

NO-A104 949

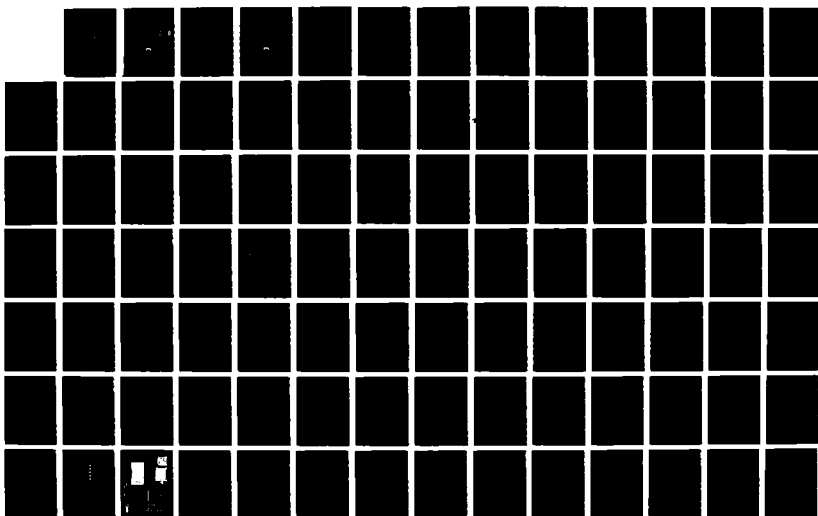
PROCEEDINGS OF THE WORKSHOP ON FUTURE DIRECTIONS IN
COMPUTER ARCHITECTURE... (U) BATTELLE COLUMBUS LABS
RESEARCH TRIANGLE PARK NC D P AGRAMAL ET AL. 30 AUG 86
ARO-06304-EL DAG29-01-D-0100

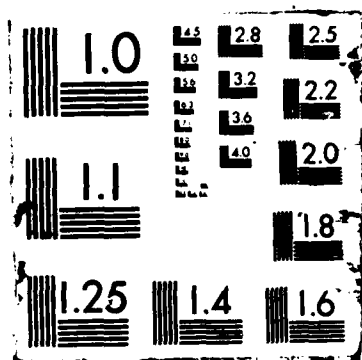
1/5

UNCLASSIFIED

F/G 12/5

NL





AD-A184 949

Proceedings

of the

Workshop

on

Future Directions in Computer Architecture and Software

May 5-7, 1986
Seabrook Island
Charleston, SC

Workshop Chairman:

Prof. Dharma P. Agrawal
Electrical & Computer Engineering
Box 7911
North Carolina State University
Raleigh, NC 27695-7911

Contract No. DAAG29-81-D-0100
Delivery Order 1974
Scientific Services Program

DTIC
ELECTE
SEP 17 1987
S C D



ARO Representative:

Dr. C. Ronald Green
Army Research Office
P.O. Box 12211
Research Triangle Park
North Carolina 27709-2211

The views, opinions, and/or findings contained in this report are those of the author(s) and should not be constructed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION			1b. RESTRICTIVE MARKINGS			
2a. SECURITY CLASSIFICATION AUTHORITY			3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.			
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE						
4. PERFORMING ORGANIZATION REPORT NUMBER(S)			5. MONITORING ORGANIZATION REPORT NUMBER(S) 86304-EL			
6a. NAME OF PERFORMING ORGANIZATION Performed by Author(s): under subcontract to: Battelle Columbus Laboratories		6b. OFFICE SYMBOL (if applicable)		7a. NAME OF MONITORING ORGANIZATION U. S. Army Research Office		
6c. ADDRESS (City, State, and ZIP Code) 200 Park Drive, P.O. Box 12297 Research Triangle Park, NC 27709				7b. ADDRESS (City, State, and ZIP Code) P. O. Box 12211 Research Triangle Park, NC 27709-2211		
8a. NAME OF FUNDING / SPONSORING ORGANIZATION U. S. Army Research Office		8b. OFFICE SYMBOL (if applicable)		9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER DAAG29-81-D-0100		
8c. ADDRESS (City, State, and ZIP Code) P. O. Box 12211 Research Triangle Park, NC 27709-2211				10. SOURCE OF FUNDING NUMBERS		
				PROGRAM ELEMENT NO. 1974	PROJECT NO.	TASK NO.
11. TITLE (Include Security Classification) Future Directions in Computer Architecture and Software						
12. PERSONAL AUTHOR(S) Editor Dharmar P. Agrawal						
13a. TYPE OF REPORT WORKSHOP		13b. TIME COVERED FROM _____ TO _____		14. DATE OF REPORT (Year, Month, Day) 86/8/30		15. PAGE COUNT 421
16. SUPPLEMENTARY NOTATION This Scientific Services Program task was requested and funded by the Monitoring Agency.						
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) Parallel Architectures, Granularity, Off-the-shelf versus new chips, mapping algorithms to architectures, distributed databases, distributed operating systems, reusable and retargetable software			
FIELD	GROUP	SUB-GROUP				
19. ABSTRACT (Continue on reverse if necessary and identify by block number) Topics covered (19) were discussed and comments regarding future directions were accumulated.						
20. DISTRIBUTION / AVAILABILITY OF ABSTRACT <input type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS				21. ABSTRACT SECURITY CLASSIFICATION		
22a. NAME OF RESPONSIBLE INDIVIDUAL				22b. TELEPHONE (Include Area Code)		22c. OFFICE SYMBOL

Proceedings

of the

Workshop

on

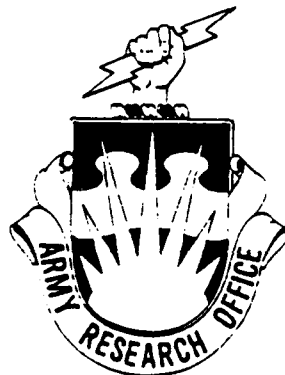
Future Directions in Computer Architecture and Software

May 5-7, 1986
Seabrook Island
Charleston, SC

Workshop Chairman:

Prof. Dharma P. Agrawal
Electrical & Computer Engineering
Box 7911
North Carolina State University
Raleigh, NC 27695-7911

Contract No. DAAG29-81-D-0100
Delivery Order 1974
Scientific Services Program



ARO Representative:

Dr. C. Ronald Green
Army Research Office
P.O. Box 12211
Research Triangle Park
North Carolina 27709-2211



Accession For	
NTIS CRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

The views, opinions, and/or findings contained in this report are those of the author(s) and should not be constructed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

PREFACE

→ The U.S. Army Research Office (ARO) sponsored Workshop on "Future Directions in Computer Architecture AND software" convened on Monday, 5 May 1986 at the Seabrook Conference Resort in Charleston, SC. LtC. Frank Ward, Office of Under Secretary of Defense, delivered the keynote address which contained thought provoking examples and a challenge on the technological issues facing the computer hardware and software technical community. A distinguished speaker lecture was presented by Dr. Stephen F. Lundstrom, Vice-President and Program Director of Parallel Processing at Microelectronics and Computer Technology Corporation (MCC).

The workshop was attended by 95 professionals: five from government agencies, 22 from private industry, and the remainder from various academic institutions. Over sixty formal position papers were organized in 19 serial sessions for the three day workshop. The final afternoon of the workshop was utilized to establish definitive insights into the perceived future research thrusts in computer hardware and software.

→ Topics presented in the position papers varied from VLSI design to retargetable software and from database to graphics. A list of the topics discussed at the workshop includes:

- Instruction Set Considerations
- Custom Chips
- Memory Hierarchy and Parallel Architecture
- Interconnection Strategies
- Reconfiguration Strategies
- Granularity issues
- Mapping Algorithms and Task Assignment
- Reusable and Retargetable Software
- Distributed Operating Systems
- Concurrency Control
- MIMO Parallelism and Support
- Distributed Computing Systems
- Architecture and Software Issues
- Logic and Functional Programming
- VLSI Related Issues
- Applicative Language and Data Flow Techniques

The objectives for this workshop were to inform the research community of the research opportunities offered by the Army Research office (ARO) and to set general directions and priorities for future research sponsored by ARO. These recommended research thrusts are included as part of these proceedings and should prove helpful in focusing future researchers on research topics of special interest to the Army.

C. Ronald Green
Workshop Sponsor
Army Research Office
Electronics Division

Contents

Preface - <i>C. Ronald Green</i>	i
1. Keynote Address - <i>Frank Ward</i>	1
2. Instruction Set Considerations	
I. Some Further Observations about Reduced Instruction Set Computers - <i>Douglas Jensen</i>	17
II. An Empirical Analysis of the Lillith Instruction Set - <i>Robert Cook</i>	21
III. High-Performance CPU Implementation of the Nebula Instruction Set Architecture - <i>Raymond Cheng</i>	29
3. Custom Chips	
I. The Arithmetic Cube and It's Associated Algorithm - <i>Mary Irwin, Robert Owens</i>	38
II. The Design of Ultra High-Speed Numeric Intensive Systems (having Fault Tolerant and Load Balancing Capabilities) - <i>F. Taylor</i>	48
III. Hardware Implementation of Special Functions: An Application for Combat Simulation - <i>John Gilmer, Jr.</i>	55
4. Memory Hierarchy and Parallel Architecture	
I. Research and Development Trends for Memory Hierarchies - <i>Alan J. Smith</i>	62
II. Signal and Data Processing for IR Sensors - <i>Doyce Satterfield, R.C. Styles</i>	72
III. On Parallelism in Software/Hardware Design Tools - <i>P.A. Subrahmanyam</i>	78
5. Interconnection Strategies	
I. Role of Broadcasting in Multiprocessor Systems - <i>Binay Sugla, Sudhir Ahuja</i>	86
II. Image Texture Classification with an Optical Crossbar Interconnected Processor - <i>Alastair McAulay</i>	90
III. Multiple-Bus Interconnection for Future Multiprocessor Systems - <i>L.N. Bhuyan</i>	98
6. Reconfiguration Strategies	
I. Automatically Reconfigurable Computer Architecture - <i>F. Gail Gray</i>	103
II. Implementation of a Buddy Failure Recovery Concept in the NOVAC System - <i>B.C. Desai</i>	114
III. Clouds: A Support Architecture for Fault Tolerant, Distributed Systems - <i>Partha Dasgupta, R.J. LeBlanc, Jr.</i>	122

7. Grainularity Issues	
I. A Large-Grain Dataflow Architecture - <i>Ian Kaplan</i>	131
II. SISAL: Initial MIMD Performance Results - <i>R.R. Oldehoeft, D.C. Cann, S.J. Allan</i>	139
8. Mapping Algorithm and Task Assignment	
I. The Mapping of Parallel Algorithms to Reconfigurable Parallel Architecture - <i>Leah H. Jamieson, Howard J. Siegel, Edward Delp, Andrew Whinston</i>	147
II. A Distributed System Architecture Based on Macro Dataflow Model - <i>Jane W.S. Liu, Andrew Grimshaw</i>	155
III. A Comparative Study of Heuristic Algorithms for Task Assignment in Distributed Computing Systems - <i>Kemal Efe</i>	163
9. Reusable and Retargetable Software	
I. Why Reusable Software Isn't - <i>William J. Tracz</i>	171
II. Towards Reusable Software Designs and Implementations - <i>Ralph E. Johnson, Simon Kaplan</i>	178
III. Archotyping - A Knowledge-Base Reuse Paradigm - <i>S.M. Przybylinski</i>	186
10. Distributed Operating Systems	
I. Distributed Control of Large Parallel Computers - <i>Larry Wittie</i>	195
II. The Design of Load Balancing Strategies for Distributed Systems - <i>Rafael Alonso</i>	202
III. Experiments with Parallel Program Execution on a Network of Workstations - <i>Willy Zwaenepoel</i>	208
11. Distinguished Lecture	
I. Future Directions in Computer Architecture and Software - The Year 2000 - <i>Stephen Lundstrom</i>	214
12. Concurrency Control	
I. Distributed Query Processing - Present Status and Future Direction - <i>Clement Yu, K.C. Guh</i>	220
II. A Paradigm for Concurrency Control Performance Evaluation - <i>A.A. Helal, Ahmed Elmagarmid, A.R. Hurson</i>	228
III. Concurrency Control and Reliability in Replicated Database Systems - <i>Mukesh Singhal</i>	236

13. MIMD Parallelism and Support	
I. Predicate Analysis for Parallel Program Generation - <i>B. Szymanski</i>	245
II. An Investigation of Parallelism in Rule Based Systems - <i>Steven Raney, David A. Marshall</i>	253
III. Software Support for Heterogeneous Machines - <i>Mario Barbacci</i>	261
14. A. Interconnection Strategies/Distributed Computing Systems	
I. The Case for a Shared Address Space - <i>Edward Gehringer</i>	270
II. The Next Generation of Hypercube Computers - <i>Trevor Mudge</i>	273
III. A Parallel Computer Based on Cube Connected Cycles - <i>Moon J. Chung,</i> <i>E.J. Toy, A.A. Lobo</i>	276
IV. An Analysis of the Reliability of Tree-Structured Interconnection Networks - <i>Ravi Mehrotra, Edward Gehringer</i>	279
V. Performance Evaluation of a Loop Interconnection Structure - <i>Edward Page</i>	284
VI. On the Number of Task Assignments in Distributed Computing Systems - <i>Kang G. Shin</i>	287
B. Architecture and Software Issues	
I. Reuse: A Reliable Unified Service Environment for Distributed Systems - <i>Lionel M. Ni, Thomas Gendreau</i>	290
II. Aspects of a Multiprocessor Architecture - <i>R. Pose, M.S. Anderson,</i> <i>C.S. Wallace</i>	293
III. Directions in Computer Graphics Architecture - <i>John Staudhammer</i>	296
IV. Developing a Standard Taxonomy of Software Engineering Standards - <i>John W. Fendrich</i>	299
V. SAGE: The Clemson University Systolic Array Generator - <i>Roy Pargas,</i> <i>Keith R. Allen</i>	301
VI. Strategies for Concurrent Processing of Complex Algorithm - <i>John Stoughton, Roland R. Mielke</i>	304
VII. Portable (and Disposable) Interpreters - <i>F. Testard-Vaillant</i>	307
C. Logic and Functional Programming	
I. A Ruled-Based Lisp Dialect Translator Using Paramodulation - <i>Michael Dowell, Yoshiyasu Takefuji</i>	316
II. Semi-Applicative Programming - <i>N.S. Sridharan</i>	319
III. Experimenting with Parallel Programming in Logic - <i>Abraham Waksman</i>	323
IV. Parallel Architecture for Logic Programming - <i>Vipin Kumar, Yow-Jian Lin</i>	326

v.	Real Time Artificial Intelligence Architecture - <i>Peter Green,</i> <i>Ronald J. Juels, William Michalson</i>	328
15.	A. Miscellaneous	
I.	A Strategy for Failure Prediction - <i>D. Andrews</i>	332
II.	Programming Language Translation for Multicomputers - <i>Jon Mauney</i>	334
III.	Resilient Procedure: A Structured Replication Approach - <i>Kwei-Jay Lin,</i> <i>Mark E. Wittle</i>	336
	B. VLSI Related Issues	
I.	Parallelism at the Microlevel: Cooperative Microcontroller Design with Real Time Considerations - <i>C.A. Papachristou</i>	340
II.	Wafer Scale Implementation of a GaAs Systolic Signal Processor Cell - <i>John F. McDonald, et al.</i>	343
III.	Design for a TPL Compiler System - A System for Retargeting High Level Language Programs - <i>S. Leong, O. Jiang, S. Jodid, P.A.D. de Maine</i>	346
IV.	A VLSI Implementable Block Oriented Data Driven Multiprocessor - <i>Behrooz Shirazi, A.R. Hurson</i>	349
v.	On Systolic Architectures for Interpolation and Integration - <i>S.I. Omar,</i> <i>G.H. Masapati</i>	353
vi.	An Applicative Programmers' Approach to Matrix Algebra, Lessons for Hardware and Software - <i>David S. Wise</i>	357
	C. Applicative Language and Data Flow Techniques	
I.	Implementing Logical Variables on a Graph Reduction Architecture - <i>Gary Lindstrom</i>	361
II.	Multi-Processor Reduction Machine Minimization - <i>Jack L. Meador,</i> <i>M.L. Manwaring</i>	364
III.	Massive Fine-Grain Parallelism in Array Computation - A Data Flow Solution - <i>Guang R. Gao</i>	367
IV.	Automated Data Flow Diagram Verification - <i>Reva Friedman, Waldo Kabat,</i> <i>W. Kozaczynski</i>	370
v.	Finely Grained Parallelism in an Applicative Architecture - <i>John T. O'Donnell</i>	372
16.	Recommendations	375
17.	List of Attendees	406
18.	Author Index	413

Session 1: Keynote Address

by

Frank Ward

**Office of the Secretary of
Defense**

Chairperson:

C. Ronald Green

Army Research Office

THANK YOU VERY MUCH FOR THE INVITATION TO SPEAK HERE TODAY. IT IS AN HONOR TO ADDRESS A SELECT GROUP SUCH AS THIS ONE. I AM ESPECIALLY PLEASED SINCE I CAN APPROACH THE TOPIC OF THIS CONFERENCE, FUTURE DIRECTIONS IN COMPUTER SOFTWARE AND ARCHITECTURE FROM TWO PERSPECTIVES. FIRST, I CAN SEE THE IMPORTANCE OF THE CONFERENCE FROM THE VIEWPOINT OF THE DEPARTMENT OF DEFENSE, SINCE MY BOSS IS THE DIRECTOR OF COMPUTER SOFTWARE AND SYSTEMS FOR THE DEPARTMENT. SECONDLY, AND MOST IMPORTANT, I CAN TAKE THE VIEW OF AN ARMY OFFICER, KNOWING THAT THE ARMY AND THE OTHER MILITARY SERVICES ARE TOTALLY DEPENDENT ON COMPUTERS TO ACCOMPLISH OUR COMBAT MISSION. THERE ARE TWO VERY IMPORTANT OUTPUTS EXPECTED FROM THIS CONFERENCE. THE FIRST IS INTANGIBLE. IT IS THE SHARING OF KNOWLEDGE BETWEEN THE PARTICIPANTS WHICH WILL ADVANCE THE GROUP'S LEARNING AND ABILITY. THE SECOND IS A SET OF GROUP RECOMMENDATIONS ON RESEARCH TO BE PURSUED IN THE AREAS WE'VE TALKED ABOUT IN THE CONFERENCE.

THE SCHEDULE FOR THE WORKSHOP IS AN AMBITIOUS ONE. WE HAVE ELEVEN SESSIONS ADDRESSING IMPORTANT TOPICS SUCH AS DISTRIBUTED AND PARALLEL HARDWARE AND SOFTWARE, REUSABLE SOFTWARE, INTERCONNECTION STRATEGIES, CUSTOM HARDWARE, AND AN ISSUE NEAR AND DEAR TO MY HEART, INSTRUCTION SET ARCHITECTURES.

WE ARE FORTUNATE TO HAVE AS OUR DISTINGUISHED SPEAKER DR. STEVE LUNDSTROM, VICE PRESIDENT OF THE MICROELECTRONICS AND COMPUTING CONSORTIUM. HE AND HIS COLLEAGUES ARE CONDUCTING FIRST RATE RESEARCH IN AREAS IMPORTANT TO THIS CONFERENCE.

I AM HERE ON BEHALF OF MY BOSS, DR. EDWARD LIEBLEIN, WHO FULLY INTENDED TO ACCEPT YOUR INVITATION, BUT WAS PREEMPTED BY A SHORT FUSED COMMITMENT AT THE LAST MOMENT. HE SENDS HIS REGRETS. HE AND THE DEPUTY UNDER SECRETARY OF DEFENSE FOR RESEARCH AND ADVANCED TECHNOLOGY, APPRECIATE THE OPPORTUNITY TO SHARE OSD'S VIEWPOINTS WITH YOU, AND THEY UNDERSTAND THE IMPORTANCE OF COMPUTERS TO OUR MISSION TO PROTECT THE SECURITY OF THE UNITED STATES.

THIS IS A VERY IMPORTANT CONFERENCE BECAUSE IT ADDRESSES THE FUTURE OF OUR MILITARY USE OF COMPUTERS. YOU PROBABLY KNOW THAT VIRTUALLY EVERY SYSTEM WE PLAN TO FIELD IN THE NEXT FEW YEARS IS CRITICALLY DEPENDENT ON ONE OR MORE COMPUTERS TO GET THE JOB DONE. WE WERE WORKING ON SOME REMARKS FOR THE UNDER SECRETARY OF DEFENSE FOR RESEARCH AND ENGINEERING, DR. DON HICKS, AND WE CAST ABOUT FOR A SYSTEM THAT DID NOT USE A COMPUTER. WE FOUND ONE, AND IT WAS THE M-16 RIFLE. OF COURSE, THAT RIFLE WAS PROBABLY DESIGNED WITH THE HELP OF A COMPUTER, SO IT MAY NOT BE A VALID EXAMPLE. AS A MEASURE OF JUST HOW DEPENDENT WE ARE ON COMPUTERS IN OUR MILITARY SYSTEMS, WE ESTIMATE THAT THE THREE SERVICES HAVE 185000 COMPUTERS IN SERVICE RIGHT NOW, AND THAT THE NUMBER WILL

DOUBLE IN THE NEXT FOUR YEARS. BY THE WAY, THAT NUMBER DOES NOT INCLUDE MICROPROCESSORS. THE AIR FORCE'S F-15 EAGLE FIGHTER AIRCRAFT HAS OVER 60 MICROPROCESSORS ON BOARD.

IT'S OBVIOUS THAT SINCE VIRTUALLY EVERY DEFENSE SYSTEM DEPENDS ON COMPUTERS TO GET THE JOB DONE, THEY ALSO DEPEND ON SOFTWARE. THERE HAVE BEEN GREAT ADVANCES IN COMPUTER HARDWARE, BUT THE SOFTWARE JUST HAS NOT KEPT UP. OUR SYSTEMS ARE PLAGUED WITH PROBLEMS THAT ARE ATTRIBUTED TO SOFTWARE.

MUCH OF OUR SOFTWARE HAS DEFECTS THAT ARE NOT FOUND UNTIL THE SOFTWARE IS FIELDDED, WHEN THE COST OF CORRECTING THE ERROR CAN BE 100 TIMES THAT TO CORRECT AN ERROR FOUND IN THE DESIGN OR CODING PHASE. ONE REASON FOR THIS SITUATION IS THE VERY COMPLEXITY OF THE SOFTWARE IN OUR MISSION-CRITICAL SYSTEMS. DOD SYSTEMS USE SOME OF THE MOST COMPLEX SOFTWARE EVER ATTEMPTED. IT IS VIRTUALLY IMPOSSIBLE TO RIGOROUSLY TEST EVERY SECTION OF CODE BEFORE THE SYSTEM GETS FIELDDED. MOST DOD SOFTWARE SYSTEMS ARE OVER A MILLION LINES OF SOURCE CODE. THAT MAKES IT VIRTUALLY IMPOSSIBLE FOR ANY ONE OR EVEN A SMALL GROUP OF PEOPLE TO UNDERSTAND WHAT THE SYSTEM IS DOING. OTHER PROBLEMS ARE CAUSED BY THE PREVALENCE OF ASSEMBLY LANGUAGE IN MILITARY SYSTEMS. UNTIL WE CAN GET THE BULK OF OUR SOFTWARE DONE IN A MODERN HIGH ORDER LANGUAGE THAT PROVIDES STRUCTURE AND EASE OF UNDERSTANDING, WE WILL CONTINUE TO HAVE HUGE, POORLY DOCUMENTED PROGRAMS THAT ARE FILLED WITH BUGS. MILITARY SOFTWARE IS STILL DONE IN A HIGHLY LABOR INTENSIVE WAY. UNTIL WE CAN BRING A DEGREE OF

AUTOMATION TO THE SOFTWARE DESIGN, CODING, MAINTENANCE AND MANAGEMENT PROCESSES, WE WILL CONTINUE TO LAG BEHIND IN OUR EFFORTS TO PROVIDE RELIABLE SOFTWARE TO OUR FORCES IN THE FIELD. THE PROBLEM WITH SOFTWARE PEOPLE WILL ONLY GET WORSE. THE UNITED STATES AS A WHOLE IS CURRENTLY SHORT 50-100,000 SOFTWARE PROFESSIONALS. BY 1990, THE SHORTAGE WILL GROW TO OVER 1 MILLION. WE MUST INCREASE THE PRODUCTIVITY OF THESE SCARCE PROFESSIONALS. WE HAVE ANOTHER SERIOUS PROBLEM WITH OUR SOFTWARE. IT COSTS TOO MUCH. THIS YEAR, DOD WILL SPEND ABOUT \$10 BILLION ON TACTICAL SOFTWARE ALONE. THIS DOES NOT INCLUDE THE SOFTWARE FOR OUR BUSINESS ADP SYSTEMS. WE ESTIMATE THAT BY 1990, WE WILL SPEND \$30 BILLION A YEAR ON THE SOFTWARE FOR THESE SYSTEMS. THAT'S OVER 10% OF THE ENTIRE DEFENSE BUDGET, AND WE CAN'T AFFORD IT. ANOTHER CONTRIBUTOR TO THE SOFTWARE CRISIS IS THE OLD TECHNOLOGY WE USE IN DEVELOPING AND MAINTAINING MISSION SOFTWARE. THIS OLD TECHNOLOGY IS NOT LIMITED TO THE HIGH USE OF ASSEMBLY LANGUAGE, BUT SPANS THE WHOLE SOFTWARE LIFE CYCLE.

NOW THAT I'VE GOTTEN THE GLOOM AND DOOM OFF MY CHEST, WHAT DO WE PLAN TO DO ABOUT IT? OUR RESPONSE TO THE SOFTWARE CRISIS IS THE DOD SOFTWARE INITIATIVE. THE INITIATIVE IS ORGANIZED AT THE DOD LEVEL BECAUSE THE PROBLEM COVERS ALL OF DOD. WHEN YOU CONSIDER DOD AS A WHOLE, WE ARE THE LARGEST SINGLE BUYER OF SOFTWARE IN THE WORLD, HENCE WE CAN INFLUENCE THE MARKET AND THE STATE OF PRACTICE IF WE ACT TOGETHER. WE ALSO FEEL THAT AN INITIATIVE MANAGED AT DOD WILL PROVIDE THE CRITICAL MASS OF PEOPLE AND FUNDING THAT THE PROBLEM DEMANDS. THE GOAL OF OUR

INITIATIVE IS TO DEVELOP ADVANCED METHODS, TECHNIQUES AND TOOLS TO REDUCE COSTS, SHORTEN SCHEDULES, AND IMPROVE THE RELIABILITY AND ADAPTABILITY OF FUTURE MISSION-CRITICAL SOFTWARE.

IT IS CRITICAL TO THE SUCCESS OF THE DOD SOFTWARE INITIATIVE THAT WE INVOLVE ALL OF THE SERVICES AND AGENCIES. WE WILL FAIL IF WE ALLOW THE INTERESTS OF ONE COMPONENT TO PREDOMINATE AT THE EXPENSE OF THE OTHERS. WE MUST MOTIVATE INDUSTRY TO INVEST ITS SKILLS AND FUNDS IN THE SOFTWARE INITIATIVE IF WE ARE TO HAVE A SIGNIFICANT EFFECT ON THE PROBLEMS FACING US. WE MUST ALSO COORDINATE THE SOFTWARE INITIATIVE WITH OTHER HIGH LEVEL NATIONAL INITIATIVES SUCH AS THE STRATEGIC COMPUTING PROGRAM, AND THE VHSIC PROGRAM.

THE SOFTWARE INITIATIVE HAS THREE COMPONENTS. THEY ARE THE ADA COMMON HIGH ORDER LANGUAGE EFFORT, THE SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS OR STARS PROGRAM AND THE DOD SOFTWARE ENGINEERING INSTITUTE. I'LL NOW GIVE A BRIEF OVERVIEW OF EACH.

THE ADA PROGRAM IS DOD'S EFFORT TO PROVIDE A SINGLE, POWERFUL HIGH ORDER PROGRAMMING LANGUAGE FOR ALL MISSION-CRITICAL SYSTEMS. THE NEED FOR A STANDARD LANGUAGE WAS MOTIVATED BY THE PROLIFERATION OF OVER 400 LANGUAGES AND INCOMPATIBLE DIALECTS IN USE IN OUR SYSTEMS. THIS PROLIFERATION LED A POOR SUPPORT BASE FOR FIELDED SOFTWARE DUE TO THE EXPENSE OF SUPPORT TOOLS. ADDITIONALLY, THE HIGH ORDER LANGUAGES IN USE BEFORE ADA WERE

GENERALLY UNSATISFACTORY FOR REAL TIME SYSTEMS. THE RESULT WAS HIGH COSTS, EXTENDED SCHEDULES (ONE SYSTEM HAD A 48 MONTH SCHEDULE EXTENSION DUE TO SOFTWARE), FAILURES IN FIELDED SYSTEMS, AND VERY LIMITED FLEXIBILITY. THE DEPARTMENT THEN UNDERTOOK A JOINT EFFORT TO DEVELOP REQUIREMENTS FOR A STANDARD LANGUAGE, DETERMINED THAT A NEW LANGUAGE WOULD BE NEEDED AND EMBARKED ON LANGUAGE DEVELOPMENT EFFORT. IN AN EXCEPTIONALLY OPEN DEVELOPMENT PROCESS, FOUR COMPETITIVE DESIGNS WERE NARROWED TO TWO AND THEN TO ONE BETWEEN 1977 AND 1980. IN THE PROCESS, OVER 7000 COMMENTS AND RECOMMENDATIONS WERE RECEIVED FROM SOFTWARE EXPERTS IN 15 NATIONS. THE RESULTING LANGUAGE DESIGN WAS ADOPTED AS MILITARY STANDARD 1815 AND AN AMERICAN NATIONAL STANDARDS INSTITUTE STANDARD IN JANUARY 1983. ADA HAS SOME INTERESTING SOFTWARE ENGINEERING FEATURES THAT MAKE IT ESPECIALLY USEFUL FOR LARGE SCALE, LONG LIVED, COMPLEX SYSTEMS. (REMEMBER THE AVERAGE LENGTH OF OVER A MILLION LINES OF SOURCE CODE PER SYSTEM.) IT IS ORIENTED TOWARD SPECIFICATION AND DESIGN, AND IS THEREFORE IN USE IN MANY SYSTEMS AS A PROGRAM DESIGN LANGUAGE. IT IS STRONGLY MODULAR IN NATURE AND USES STRUCTURES CALLED PACKAGES AS SOFTWARE BUILDING BLOCKS AND TASKS TO SPECIFICALLY PROMOTE CONCURRENT EXECUTION. IT HAS SPECIFIC FEATURES FOR HANDLING HARDWARE OR SOFTWARE FAILURES, AND THUS CAN BE USED TO PERMIT GRACEFUL DEGRADATION IN THE PRESENCE OF COMPONENT FAILURES, RATHER THAN TOTAL SYSTEM FAILURE. IT IS A STRONGLY TYPED LANGUAGE AND HAS EXTENSIVE FEATURES FOR COMPILE TIME CHECKING. ITS MODULES CAN BE SEPARATELY COMPILED, FACILITATING THE SOFTWARE MAINTENANCE PROCESS. IT ALSO HAS SPECIFIC FEATURES FOR INTERACTING WITH THE

TARGET COMPUTER'S HARDWARE, THUS MOVING AWAY FROM THE NEED FOR ASSEMBLY LANGUAGE.

THE UNDER SECRETARY OF DEFENSE FOR RESEARCH AND ENGINEERING HAS DIRECTED THAT ADA WILL BE USED AS THE PROGRAMMING LANGUAGE FOR ALL MISSION-CRITICAL SYSTEMS. WE CONTROL THE LANGUAGE BY SEVERAL MEANS. FIRST WE HAVE A REGISTERED TRADEMARK FOR THE NAME, AND WE ONLY ALLOW USE OF THE NAME ADA ON SOFTWARE SYSTEMS THAT HAVE PASSED THE ADA VALIDATION TESTS. THESE TESTS CONSIST OF OVER 2500 PROGRAMS, AND DETERMINE THAT A COMPILER HAS FULLY IMPLEMENTED THE STANDARD DEFINITION OF THE LANGUAGE. WE WILL PERMIT NO SUBSETS OR SUPERSSETS OF THE LANGUAGE, SINCE EITHER WOULD PUT US RIGHT BACK WHERE WE WERE WITH A HOST OF INCOMPATIBLE DIALECTS.

THE ADA PROGRAM HAS BEEN A TREMENDOUS SUCCESS AND WE ARE VERY PROUD OF ITS ACCOMPLISHMENTS. IT HAS BEEN SELECTED FOR USE IN OVER 130 DEFENSE SYSTEMS. IT HAS BEEN ADOPTED FOR USE IN THE DEFENSE SYSTEMS OF THE UNITED KINGDOM, CANADA, THE FEDERAL REPUBLIC OF GERMANY, SWEDEN, AND FOR COMMAND AND CONTROL SYSTEMS IN THE NATO ALLIANCE. IT WILL BE USED IN THE NASA SPACE STATION FLIGHT SOFTWARE, IN FAA UPGRADES, AND IN SIMILAR EFFORTS IN THE CANADIAN AVIATION ADMINISTRATION. IT WILL BE USED IN BOEING'S NEW AIRPLANE, THE 7J7. THIRTEEN DIFFERENT COMPANIES HAVE DEVELOPED 29 VALIDATED COMPILERS. SOME OF THESE COMPILERS ARE COMPETITIVE

IN TERMS OF SPEED AND CODE EXPANSION RATIOS TO COMPILERS FOR MATURE LANGUAGES. THERE ARE FIVE COMPILER VALIDATION FACILITIES, TWO IN THIS COUNTRY AND THREE IN EUROPE.

EXPERIMENTAL RESULTS FROM USE OF THE LANGUAGE TO RECODE EXISTING PROGRAMS HAVE BEEN VERY ENCOURAGING. THE FLIGHT CONTROL PROGRAMS FOR THE F-15 EAGLE AND F-20 TIGERSHARK FIGHTER AIRCRAFT WERE RECODED IN ADA, WITH DRAMATIC REDUCTIONS IN SOURCE CODE SIZE, AND TEST MISSIONS SUCCESSFULLY FLOWN. THE UNITREP PROGRAM FROM THE WORLD WIDE MILITARY COMMAND AND CONTROL SYSTEM WAS RECODED FROM THE ORIGINAL COBOL INTO ADA. THE ORIGINAL 60000 LINES OF COBOL CODE WERE REDUCED TOO 5000 LINES OF ADA. THE STRATEGIC AIR COMMAND'S MOBILE INFORMATION MANAGEMENT SYSTEM WAS RECODED FROM JOVIAL INTO ADA WITH A REDUCTION FROM 130,000 LINES TO 8100 LINES OF CODE. AN INTERESTING OBSERVATION IS THAT OVER HALF OF THE COBOL STATEMENTS IN THE WWMCCS PROGRAM WERE DATA CHECKING STATEMENTS THAT ADA'S STRONG TYPING ELIMINATES WITH COMPILE TIME CHECKING.

THE SECOND COMPONENT OF THE DOD SOFTWARE INITIATIVE IS THE SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS, OR STARS PROGRAM. IT SEEKS TO IMPROVE DOD'S ABILITY TO PROVIDE RELIABLE, COST EFFECTIVE MISSION-CRITICAL SOFTWARE BY AN ORDER OF MAGNITUDE. THE STARS EFFORT WILL BRING A HIGH DEGREE OF AUTOMATION TO THE SOFTWARE DEVELOPMENT, MANAGEMENT AND MAINTENANCE PROCESSES. IT WILL PROMOTE THE REUSE OF PROVEN PIECES OF CODE, IMPROVE THE PRODUCTIVITY OF THE SOFTWARE

PROFESSIONAL WORK FORCE BY AUTOMATION AND IMPROVED TRAINING. IT WILL IMPROVE THE WAY WE ACQUIRE SOFTWARE. IT WILL ALSO PROMOTE RAPID PROTOTYPING OF SOFTWARE SYSTEMS TO BRING THE USER, DESIGNER AND PROGRAMMER TOGETHER EARLY IN THE SOFTWARE LIFE CYCLE.

A KEY TO AUTOMATING THE SOFTWARE PROCESS IS THE USE OF SOFTWARE ENGINEERING ENVIRONMENTS, CALLED SOFTWARE FACTORIES BY SOME. THESE ENVIRONMENTS ARE INTEGRATED COLLECTIONS OF SOFTWARE TOOLS THAT DO MUCH OF THE WORK PREVIOUSLY DONE IN A MANUAL AND TIME CONSUMING WAY BY THE PROGRAMMER. THE ENVIRONMENTS TO BE USED FOR DEFENSE SOFTWARE MUST FACILITATE THE PORTABILITY OF SOFTWARE BETWEEN THE ENVIRONMENT USED BY THE CONTRACTOR, AND THE ENVIRONMENTS USED IN GOVERNMENT POST DEPLOYMENT SUPPORT CENTERS. WE MUST BE ABLE TO TRANSPORT MISSION SOFTWARE, SUPPORT TOOLS, AND PROGRAMMERS, OR WE WILL BE UNABLE TO MEET THE REQUIREMENTS OF THE 90'S.

ONE WAY TO ENSURE THE TRANSPORTABILITY OF SOFTWARE BETWEEN ENVIRONMENTS IS TO INSIST ON THE USE OF STANDARD ENVIRONMENTS. THE USE OF A STANDARD ENVIRONMENT HAS THE ADVANTAGE OF PROVIDING THE PORTABILITY DESIRED, BUT TENDS TO STAGNATE THE TECHNOLOGY, LOCKING YOU IN TO THE STANDARD. IT ALSO CARRIES RISKS OF HIGH COSTS, DELAYS AND POOR QUALITY. IT WILL TEND TO DISCOURAGE INDUSTRY FROM INVESTING IN THE TECHNOLOGY, SINCE THEY FAR PREFER TO USE THEIR OWN PRODUCTS. THE OTHER EXTREME, THE LAISSEZ-FAIRE OR NO STANDARDS APPROACH HAS THE ADVANTAGE OF A LARGE NUMBER OF

QUALITY ENVIRONMENTS DEVELOPED BY INDUSTRY, BUT PROVIDES LIMITED TRANSPORTABILITY BETWEEN ENVIRONMENTS. PERHAPS THE BEST APPROACH IS TO STANDARDIZE ON THE INTERFACES OF THE TOOLS TO THE ENVIRONMENT, AND ON ADA AS THE BASELINE FOR THE ENVIRONMENTS. THIS PROVIDES THE TRANSPORTABILITY (THROUGH STANDARD INTERFACES), CREATES A ROBUST MARKETPLACE FOR COMPONENTS OF THE ENVIRONMENTS, FACILITATES HIGHER QUALITY PRODUCTS, REDUCES COST, PROVIDES FOR CONTINUAL UPGRADE OF THE ENVIRONMENT AND ITS COMPONENTS AND GETS THE PRODUCTS TO THE FIELD FASTER.

THE STARS PROGRAM HAS HAD SOME SUCCESS IN THE AREAS OF REUSABILITY AND SOFTWARE ACQUISITION. IN SOFTWARE REUSABILITY, WE SUPPORTED A PROGRAM CALLED COMMON ADA MISSILE PACKAGES OR CAMP. THE CAMP PEOPLE FOUND THAT THERE ARE CERTAIN PRIMITIVE FUNCTIONS PERFORMED BY ALL MISSILES. THEY THEN CODED THESE PRIMITIVES IN ADA USING STANDARD INTERFACE DEFINITIONS AND STORED THEM IN A REPOSITORY OF REUSABLE PARTS. IN SOFTWARE ACQUISITION WE HAVE PROTOTYPED AND DEMONSTRATED A SOFTWARE ACQUISITION MANAGERS WORKSTATION.

THE THIRD COMPONENT OF THE SOFTWARE INITIATIVE IS THE SOFTWARE ENGINEERING INSTITUTE. IT TAKES, ON THE AVERAGE FROM 15-20 YEARS TO GET A NEW SOFTWARE ENGINEERING CONCEPT FROM THE LABORATORY INTO USE. REASONS FOR THIS LONG PERIOD OF TRANSITION FOCUS ON RELUCTANCE TO USE NEW IDEAS, AND A DIFFICULTY IN SCALING UP THE CONCEPT FROM USE ON SMALL LABORATORY PROJECTS TO USE IN LARGE SCALE SYSTEMS. THE SOFTWARE ENGINEERING INSTITUTE WAS

CREATED TO BRING THE ABLEST PROFESSIONAL MINDS AND THE MOST EFFECTIVE TECHNOLOGY TO BEAR ON THE RAPID IMPROVEMENT OF THE QUALITY OF SOFTWARE. IT WILL ACCELERATE THE TRANSITION OF NEW TECHNOLOGY INTO USE IN DEFENSE SOFTWARE, PROMOTE THE USE OF MODERN TECHNIQUES AND METHODS, AND ESTABLISH STANDARDS OF EXCELLENCE FOR SOFTWARE ENGINEERING PRACTICE. IT IS A NEW FEDERAL CONTRACT RESEARCH CENTER, ESTABLISHED IN DECEMBER 1985 AT CARNEGIE-MELLON UNIVERSITY. IN ITS FIRST YEAR, IT HAS STAFFED UP TO 100 PROFESSIONAL STAFF MEMBERS, AND COMPLETED SIGNIFICANT PROGRAM PLANNING. IT HAS EVALUATED SEVERAL EXISTING ADA ENVIRONMENTS, COMPLETED A RIGOROUS STUDY OF RIGHTS IN DATA ISSUES, DEVELOPED A MASTER OF SOFTWARE ENGINEERING CURRICULUM, ESTABLISHED AN AFFILIATES PROGRAM WITH INDUSTRY AND HAS STARTED ON A SHOWCASE SOFTWARE ENGINEERING ENVIRONMENT.

THE DOD SOFTWARE INITIATIVE HAS A VISION OF THE FUTURE, WHICH HAS DOD MEETING ITS MISSION-CRITICAL REQUIREMENTS WITH THE NEEDED QUALITY, ON TIME, AT REASONABLE COST AND DOING SO ROUTINELY AND PREDICTABILITY. WE SEE MARKET PLACES EXISTING IN SOFTWARE TOOLS, METHODS AND ENVIRONMENTS, AND IN REUSABLE SOFTWARE COMPONENTS. DOD WILL BE AN INTELLIGENT SOFTWARE BUYER. INDUSTRY AND THE DEPARTMENT WILL HAVE THE ABILITY TO RAPIDLY ESTABLISH UP TO DATE, POWERFUL, INTEGRATED SOFTWARE ENGINEERING ENVIRONMENTS AS THEY ARE REQUIRED, AND THE COST OF MISSION-CRITICAL SOFTWARE WILL BE REDUCED BY A FACTOR OF 100.

WE ALSO FACE EQUALLY CHALLENGING PROBLEMS FOR THE FUTURE IN COMPUTER HARDWARE. WE HAVE VALID REQUIREMENTS FOR COMPUTERS THAT CAN RUN AT SPEEDS OF BILLIONS OF OPERATIONS PER SECOND, BUT FIT IN THE SAME OR SMALLER SPACES THAN TODAY'S SYSTEMS. THERE HAVE BEEN TREMENDOUS ADVANCES IN CIRCUITRY RESULTING IN ORDERS OF MAGNITUDE SPEED IN COMPUTER PERFORMANCE, BUT WE ARE APPROACHING THE BARRIER IMPOSED BY THE SPEED OF LIGHT IN SIGNAL PROPAGATION WITHOUT ACHIEVING THE SPEEDS WE'LL NEED IN THE 1990'S. THE ANSWER APPEARS TO BE PARALLEL PROCESSING, ONE OF THE IMPORTANT ISSUES ADDRESSED BY THIS CONFERENCE. HOWEVER, THERE ARE SERIOUS, OPEN ISSUES FACING US BEFORE WE CAN GET PARALLEL PROCESSING INTO WIDE USE.

ONE OF THE MOST IMPORTANT QUESTIONS IS HOW DOES ONE PROGRAM A PARALLEL MACHINE TO GET THE MOST OUT OF IT? ARE EXISTING LANGUAGES ADEQUATE? ADA HAS SPECIFIC FEATURES TO SUPPORT CONCURRENT EXECUTION BY THE USE OF TASKS AND RENDEZVOUS STATEMENTS. WILL THESE BE ADEQUATE? LISP HAS BEEN EXTENDED TO PROGRAM THE CONNECTION MACHINE, A HIGHLY PROMISING MACHINE DEVELOPED FOR DARPA THAT FEATURES 64000 PROCESSORS CONNECTED BY A DYNAMIC COMMUNICATIONS NETWORK. OTHER QUESTIONS ARE HOW DO YOU DETECT THE OPPORTUNITY FOR PARALLELISM IN AN ALGORITHM AND PARTITION IT CORRECTLY? HOW DO YOU CONNECT THE PROCESSING ELEMENTS TO PROVIDE THE NEEDED COORDINATION WITHOUT TYING DOWN THE MACHINE WITH THE COMMUNICATIONS OVERHEAD? WILL WE BE ABLE TO RETARGET EXISTING SOFTWARE TO PARALLEL MACHINES?

BEFORE WE GET TO THE TIME WHEN WE HAVE PARALLEL ARCHITECTURES IN PLACE IN OUR SYSTEMS, WE MUST BE CONCERNED WITH WHAT WE DO WITH THE ARCHITECTURES AVAILABLE TO US TODAY. WE KNOW THAT INSTRUCTION-SET ARCHITECTURE STANDARDIZATION PROVIDES FOR COMMONALITY AND EASE OF REPAIR IN THE FIELD, BUT IT IS POLITICALLY UNPOPULAR BECAUSE IT IS PERCEIVED AS DISCOURAGING COMPETITION. IS THERE A WAY TO GET THE BENEFITS OF INSTRUCTION-SET STANDARDIZATION WITHOUT PAYING THE PENALTIES? ON THE TECHNICAL SIDE, WHAT DO WE DO ABOUT OUR ARCHITECTURES? IS THE REDUCED INSTRUCTION-SET COMPUTER OR RISC THE WAY TO GO? I REMEMBER RECEIVING A NASTY SHOCK WHEN THE ADA COMPILER WE WERE FOOLING AROUND WITH ON THE NEBULA ARCHITECTURE WE USED ON THE MILITARY COMPUTER FAMILY PROJECT ABSOLUTELY REFUSED TO GENERATE THE BEAUTIFUL INSTRUCTIONS I HAD INSISTED ON INCLUDING, BUT CHOSE THE PLAIN OLD SIMPLE ONES INSTEAD. WE HAVE A SESSION ON INSTRUCTION SETS IN THE CONFERENCE, AND I LOOK FORWARD TO PARTICIPATING.

THIS CONFERENCE HAS AN OPPORTUNITY TO ADDRESS THE GREATEST CHALLENGES WE FACE IN PROVIDING OUR MILITARY FORCES THE CAPABILITY TO OVERCOME AN ENEMY WHO OUTNUMBERS US AT LEAST 10 TO ONE IN ALL THE AREAS THAT MATTER. I CAN TELL YOU AS AN ARMY OFFICER THAT WE WILL ONLY PREVAIL IF WE CAN DO MORE THAN TALK ABOUT COMBAT MULTIPLIERS. OUR NATIONAL SECURITY DEPENDS ON PEOPLE LIKE YOU WHO CAN BRING OUR GREAT RESOURCES IN TECHNOLOGY TO BEAR ON THE PROBLEMS WE FACE.

THANK YOU AGAIN FOR THE OPPORTUNITY TO SPEAK TO YOU. I'M
LOOKING FORWARD TO SHARING A SUCCESSFUL CONFERENCE WITH YOU.
THANK YOU VERY MUCH. ARE THERE ANY QUESTIONS?

Session 2: Instruction Set Considerations

Chairperson: C. A. Papachristou
Case Western Reserve University

Some Further Observations about Reduced Instruction Set Computers

*A Position Paper for the Army Research Office Workshop on
Future Directions in Computer Architecture,
April 1986*

E. Douglas Jensen

COMPUTER SCIENCE DEPARTMENT
ELECTRICAL AND COMPUTER ENGINEERING DEPARTMENT
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PA 15213
412-268-2574

In previous publications (e.g., Colwell et al.¹), we have attempted to present a relatively impartial view of the "Reduced Instruction Set Computer (RISC)/Complex Instruction Set Computer (CISC)" debate. This abstract summarizes a few additional observations about the RISC/CISC controversy, with the objective of stimulating some thought in an area where more would clearly help.

The RISC design philosophy includes some significant and potentially valuable challenges to many of the implicit assumptions which have guided computer design for many years. Examples of these assumptions and challenges include:

- 'If some of the instructions are best implemented in microcode, then *all* of them are or at least can reasonably be.' Hard-core RISC people argue that all instructions should execute in a single cycle and thus there should be no microcode (unless you want to think of the machine language as being elementary microcode stored in main memory). A non-dogmatic view would be that *vertical* microcode is frequently no longer cost-effective for simple instructions, but that in many cases multicycle instructions are appropriate, perhaps some microcoded and some hardwired, and that complex instructions (whether microcoded or hardwired) need not slow down the execution of simple ones.
- 'The dichotomy between instruction set *architecture* and its *implementation* (which was one of the important contributions of the IBM System/360 family) is invariably the correct design style.' RISC designs deliberately discard this distinction — for example, making implementation mechanisms such as caches and pipelines explicitly visible and controlled at the instruction set architecture level. It is probably clear to everyone that there are certain performance advantages to be gained by this, but it is not obvious that these always outweigh the disadvantages. In a world where compatibility of system and application software is a mainstay, and a family of compatible machines is the primary vehicle for accommodating a wide range of cost/performance requirements, and any par-

¹Colwell, Robert P., Charles Y. Hitchcock III, E. Douglas Jensen, Charles Kollar, and H. Brinkley Sprunt, *Computers, Complexity, and Controversy, Computer*, September 1985, IEEE.

ticular machine is expected to perform well on a wide variety of commercial and scientific applications, the RISC approach of creating every computer architecture and implementation anew has problems. It is worth noting that the DoD effort to have several contractors competitively develop military computers based on the MIPS design includes the definition of a new level of standardization: an intermediate language as the target for all compilation, which is then translated into machine code performing low level resource management (e.g., pipeline scheduling, register allocation, cache control) optimally for each different MIPS implementation.

Such self-examination brought on in the computer systems design community by the RISC school of thought is refreshing and welcome. However, scientific objectivity and rationality have been all but swept away in the deluge of public relations fanfare generated lately by marketing, media, and even engineering advocates of RISC's (for various electro-political reasons). The CISC camp has allowed this to be a rather one-sided debate by largely maintaining its unfortunate historic low visibility of rationale, and by having a less coherent position not readily articulated in slogans. Some activists in the RISC movement apparently are deliberately attempting to polarize the situation, but we believe that there is a silent majority of computer systems designers who are wisely and impartially awaiting sufficient credible evidence before adopting positions appropriate for their circumstances inbetween the endpoints of this artificial RISC/CISC dichotomy.

The theologically purist RISC researchers have put forth not only some stimulating departures from orthodoxy, but also some misleading assertions about both their approaches and their results. Many RISC study publications have left much to be desired in scientific and engineering credibility, perhaps constituting a local minimum in the area of computer architecture; even if all their assertions were eventually to be proven totally correct, that would not excuse the poor quality of investigation and reporting. It is peculiar that high quality products like the Encyclopedia Britannica and Kirby vacuum cleaners are marketed using questionable tactics; the most skeptical observers no doubt believe that the RISC viewpoint deserves better treatment from its proponents and the media.

A few computer corporations are employing a small subset of carefully selected RISC perspectives in what, for a variety of technical and nontechnical reasons, may eventually prove to be technically and economically viable products. Then both the researchers and the companies make exaggerated claims that the work of each validates that of the other.

It is revealing to compare: the Stanford MIPS with its MIPS Computer Corp. and DoD contractor (Honeywell, RCA, Rockwell) counterparts; the IBM 801 with the RT PC and ROMP chip inside; the Berkeley RISC I and II with any commercially available RISC products such as the HP Spectrum; and the marketing hype-"RISC" systems from Ridge, Pyramid, Convex, Sperry, Harris, and others with any of the three pioneering RISC research machines.

Some of the central tenets of the RISC dogma are being reflected in few if any products (unsurprisingly, since little conclusive evidence for their efficacy has yet appeared, at least in public). One of the positive contributions of RISC work is renewed attention to the performance of the most frequently executed instructions, which tend to be simple register-to-register operations; but it is an unjustified extremist position that only those instructions should be included. It is not intrinsically or even obviously true that single-cycle execution or frequency of use are necessarily the (much less the *only*) proper criteria for inclusion in an instruction set. Frequency of use alone does not determine throughput — an infrequently used but very lengthy instruction can become the performance bottleneck (*cf* the DEC VAX MOVC). Nor does frequency of use alone determine response time — it may be critical that an infrequently used instruction execute quickly when needed (e.g., in the event of a reactor overtemperature alarm). Instruction *importance* should determine what is optimized in the instruction set architecture, even though it is harder to measure and context-dependent (not to mention more difficult to make simplistic generalizations about). Instructions should be included as justified by cost-effectiveness tradeoffs to meet current system requirements — for example: performance (e.g., algorithmic specialization, not only for the intended application² but also for the system, such as object invocation, interprocess communication, real-time scheduling); fault tolerance (e.g., support for atomic transactions, replication); software compatibility.

Most, if not all, of the new commercial computer products claimed to be "RISC's" incorporate a (sometimes considerable) number of less frequently used instructions — directly, or indirectly (*cf* the Fairchild Clipper "macroinstruction unit"), or via "extended function unit" and co-processor interfaces (*cf* the HP Spectrum machines). Such interfaces lead in the direction of the tail wagging the dog, where the best approach is for there to be a RISC co-processor in a CISC.

Some of the most important design features in the alleged RISC products have little or nothing to do with RISC's and obviously apply just as well to, and have already been used in, CISC's. Examples of these from the RISC literature include: register organization (e.g., overlapped register windows); careful assignment of complexity across different system levels; re-use of information; use of storage hierarchies; re-targetable operating systems (e.g., UNIX); simple instructions execute quickly; operating system involvement in pipeline and cache management; a priori (i.e., compiler) pipeline scheduling. However, as mentioned above, the RISC philosophy has indeed revitalized some of these topics.

The RISC school of thought has interesting dependencies on semiconductor technology. For example, it would appear to be ideal for the current low density available in GaAs; but RISC single cycle

²*cf* the SDIO Eastport Group report recommendation for higher level, algorithmically specialized, non-numeric functions

instructions (although the published, and expected, averages are in the *several* cycle range due to branches, cache misses, etc.) place excessive demands on the scarce processor/memory bandwidth, exacerbated in microprocessors by pin limitations at that interface. Since the RISC operators and operands are so trivial, keeping the processor busy becomes more difficult as it increases in speed. The number of devices possible and sufficient for a RISC processor chip are only adequate to implement a RAM chip which is so small that many of them are required in a cache large enough to keep up with the processor. While a system may be able to afford one expensive GaAs processor chip, it may not be able to afford many expensive GaAs cache RAM chips. As density rises, more cache can be migrated on-chip, but probably never enough — consider what would be needed to support a 500 MHz RISC processor.

Many of the fans and media publicists on the RISC bandwagon tend to take a sadly binary view of this controversy: you are either for RISC's or against them, there's no room for thoughtful skepticism and a rational middle ground. This attitude was expressed by overzealous political activists in the 1970's as "Either you are part of the solution or you are part of the problem". Religions, especially the most fundamentalist ones, have always insisted that you are either at peace or at war with their deity. All three of these groups erroneously believe that this author is on the wrong side of their causes. I agree with H.L. Mencken on the following (among other things):

"For every complex problem
there is a simple solution...

and it doesn't work."

An Empirical Analysis of The Lilith Instruction Set

Robert P. Cook
Department of Computer Science
University of Virginia
Charlottesville, VA 22903

ABSTRACT: We describe a static analysis of the instructions used to implement all the system software on the Lilith computer. The results are compared with a similar analysis performed on the Mesa instruction set architecture. The Lilith.mmp project is also described together with some of the experiments to be performed as part of the design effort.

KEY WORDS AND PHRASES: BISC Architecture, Tightly-coupled Multiprocessor, Instruction Set Design, Lilith, Modula-2, Stack Machine, Empirical Analysis.

1. Introduction

The Lilith.mmp(multiple modula-2 processors) project is centered around extensions to the Lilith computer[1,2], designed and implemented by Professor Niklaus Wirth[2] at the Swiss Federal Institute of Technology-Zurich (ETH). Wirth has refined the Lilith into a powerful single-user workstation. A network of over eighty Liliths is now in use at the ETH for research and teaching purposes.

As a prerequisite to the design of a second version of the Lilith architecture, we have undertaken an analysis of the effectiveness of the current instruction set. This paper describes the results of the first phase of that effort, which is a static analysis of the current Lilith software environment. This work is part of a continuing set of experiments[3-5] with problem-oriented languages and language-oriented architectures. The data serves as an independent corroboration of the experiments conducted at Xerox PARC for the Mesa instruction set[6]. In addition, information is provided to guide architects of future high-level language machines. Finally, some of the experiments proposed for the Lilith.mmp project are discussed.

The code analyzed consisted of the Medos-2 operating system, the Modula-2 compiler, and all other system software, which included text editors, document processors, window packages, the I/O library, and numerous other modules. The software was composed from 180 modules with 2,236 procedures, comprised of 146,293 instructions.

We point out that a static analysis has the most impact on reducing the size of a program's object code, while dynamic statistics help most in the area of execution speed. For example, the BitBlockTransfer(BBLT) instruction that manipulates bit maps was only used 14 times; however, one has only to use the Lilith's windowing package to appreciate the advantages of including BBLT in the instruction set.

2. The Lilith Architecture

The Lilith was designed[2] to support the Modula-2 programming language; in fact, all programming on the Lilith is in Modula-2. As a result, the hardware organization is optimized to support frequently occurring operations in the language. The Lilith is a 16-bit architecture; physical memory can be of an arbitrary size but each process' address space is restricted to 64K words. A 16-word stack is used for expression evaluation and argument passing. The evaluation stack is not checked for overflow as the compiler is expected to generate code in such a way that overflow is avoided. Figure 1 illustrates the format of the hardware state vector as well as the layout of modules, module-global data, code, and the activation record(frame) stack.

The address of the currently executing Lilith process is contained in the P register. A process is usually stored as a sequential image. The first record in a process' image is used as a save area (when the process is not executing) for the base register portion of the hardware state vector. The "in use" registers in the evaluation stack are saved on the top of the activation record(procedure frame) stack together with a count of their number. The activation record stack

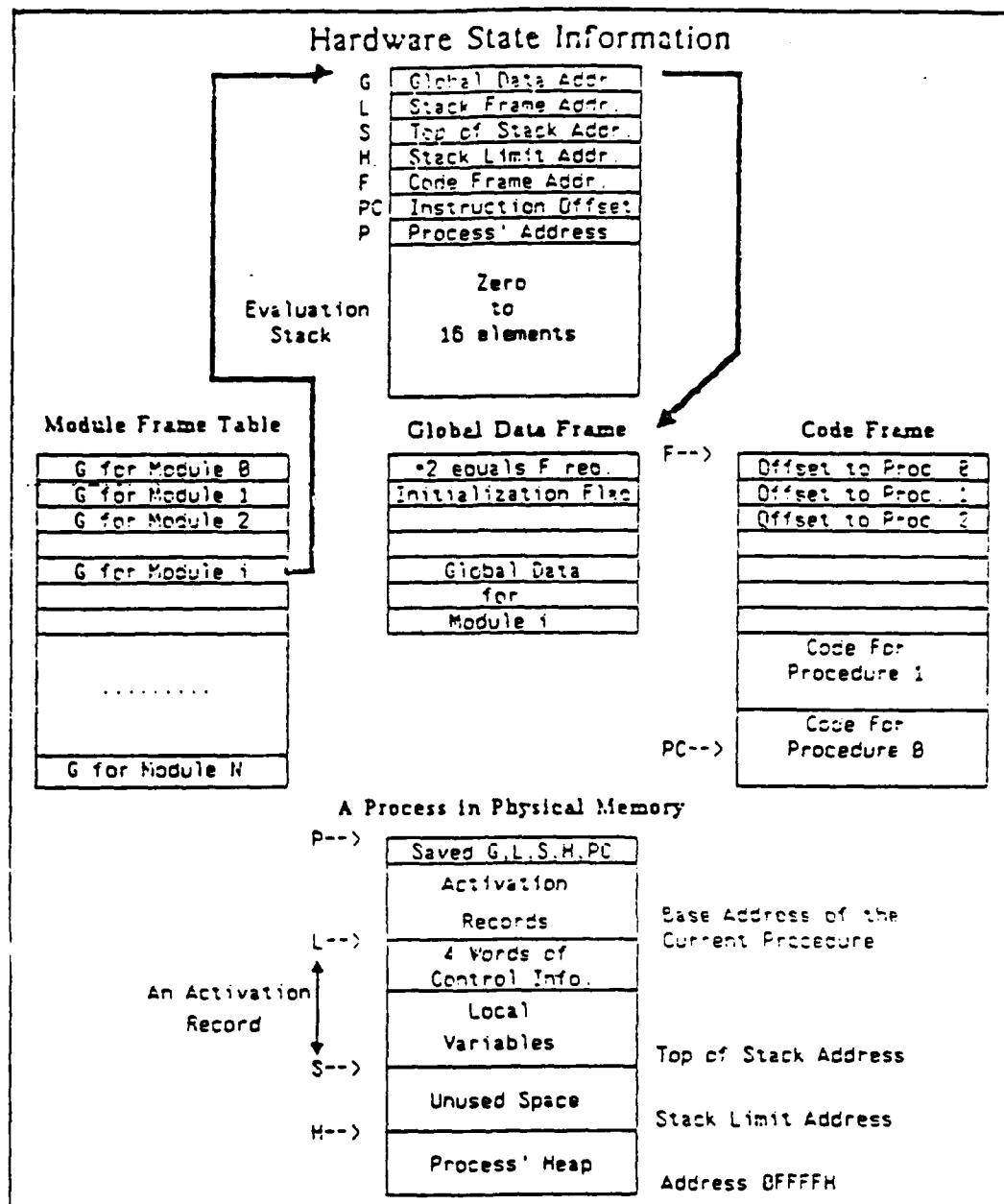


Figure 1

follows the save area in a process' image.

When a process is executing, the L register contains the address of the activation record for the current procedure, the S register points to the stack top for that execution, and the H register determines the stack limit address. The L register is also used as a base register for accesses to the local variables of a procedure.

A procedure variable on the Lilith is represented as an 8-bit module number and an 8-bit procedure number within the selected module. On a procedure invocation, the module number is used as an index into the module frame table, which is a vector of relocation bases for modules. The module frame table is typically shared among all processes. The address in the module frame table entry points to the global data frame of the selected module and is used to initialize the G register. The G register is used as a base register for accesses to the global variables of a module.

The first word of the global data frame contains the address of the code frame (instructions) for a module. In order to maximize the use of the data memory (the lower 64K words), the value is multiplied by two to yield a 17-bit address; thus, code frames can be relocated anywhere in the

first 128K words of memory. The resulting address is then loaded into the F register, which serves as the base register to the instructions for a module.

On a procedure call, the 8-bit module number is used to select the global data and code frames; next, the 8-bit procedure number is used to index an offset vector that is stored at the beginning of the code frame. The offset corresponding to the procedure number is then used to initialize the program counter register(PC). By convention, procedure zero holds the instructions for the initialization code of a module. Also by convention, procedure zero is expected to set the second word of the global data frame to TRUE to indicate that module initialization is in progress.

In summary, there are three important base registers on the Lilith--G, L, and F. G locates a module's global variables, L locates a procedure's local variables, and F locates a module's instructions.

2.1 The Lilith instruction set

The Lilith uses the 16-word evaluation stack to compute expressions. The data types that are supported include BOOLEAN, BITSET, CARDINAL, INTEGER, CHAR, and REAL. Figure 2 illustrates the instruction formats and addressing modes for the Lilith. Only 50 instructions out of the 256 possible opcodes occupy more than a single byte. Of the 206 "short" instructions, 115 are of the second format; that is, a 4-bit opcode and a 4-bit selection value. The format-two instructions were allocated to what was assumed to be the most frequently occurring operations; e.g. immediate(-1..15), load/store local(0..11), load/store global(2..15), load/store evaluation stack indirect(0..15), and local procedure calls(1..15).

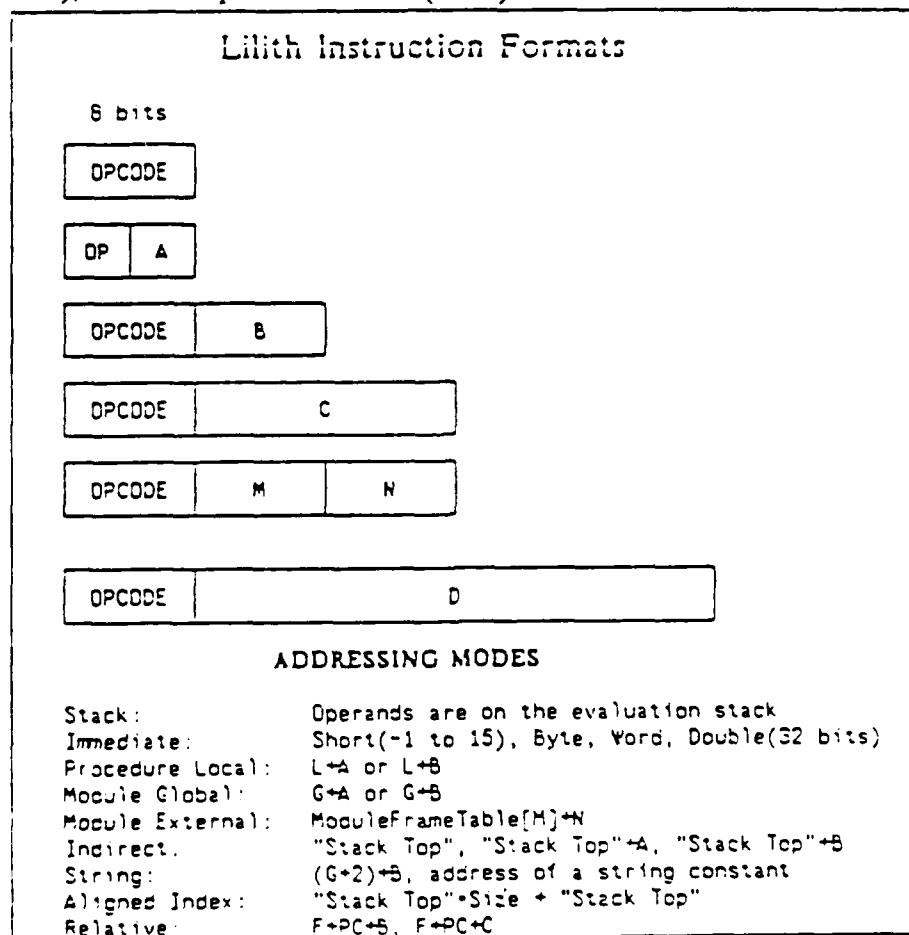


Figure 2

If the instructions were reasonably matched to Modula-2's needs, one would expect compact program representations. This expectation certainly seems to be realized given the small size of the system's software. Also, Wirth[8] retargeted the code generator of the Modula-2 compiler for both the National Semiconductor 32000 and the Motorola 68000.

"The compiler program is 14% longer for NS and 56% longer for MC than for Lilith. If we consider the code generator parts only, the respective figures are 37% and 154%. But most disappointingly, the reward for all these efforts and expenses appears as negative: for the same programs, the compiled code is about 50% longer for NS and 130% longer for MC than for Lilith. The cumulative effect of having a more complicated compiling algorithm applied to a less effective architecture results in the compiler for the NS being 1.8 times more voluminous than that for Lilith, whereas the compiler for the MC is 3.3 times as long."

3. A Comparison With The Mesa Architecture

The Lilith architecture was inspired by the Mesa instruction set architecture for the XEROX PARC Alto, which was later refined in designing the Dorado, and is currently being refined again as part of the Dragon project[7]. The comparisons, described later in the paper, are with the analysis by Sweet and Sandman[6] of all the software running under Pilot, the Mesa operating system. They converted the existing object code into a canonical form. This included breaking the code into straight line sequences, and undoing most peephole optimizations. The intent was to remove any biases introduced by the compiler's code generation strategies. Their sampling resulted in 2.5 million bytes of normalized instructions.

The instruction sets for the Lilith and Mesa architectures are very similar. For example, both use the G, L and F registers in an identical fashion and both maintain a module frame table. There are a few differences, however. Mesa stores constants in a procedure's code frame; this is not possible on the Lilith. Another difference arises in references to symbols imported from another module.

To refer to external variables or procedures on the Lilith, an instruction must include the symbol's module number and an offset. The address is then generated by indexing the module frame table. In the Mesa architecture, each module has a link area, which is initialized with the module number and offset of external symbols. As a result, external procedure calls require one more level of indirection than on the Lilith. However, an external reference on the Lilith is encoded as a three-byte instruction; whereas on the Alto, it would normally occupy only a single byte.

Figure 3 compares the opcode distribution on the Lilith with that of the Mesa architecture. The categories are derived from the normalized data presented by Sweet and Sandman[6]. The data provides a good illustration of how code generation strategies and language usage can affect opcode statistics. With respect to the Mesa statistics, we omitted JNE/JE(3.94%) because there was no direct comparison with the Lilith's JPFC/BC(4.52%). There were also some frequently-occurring Lilith instructions(CHKZ(check zero-origin array bound), ENTR(enter procedure), GBL(get L from nested procedure)) that did not have Mesa counterparts. Even though the match is not exact, the sets of instructions are quite comparable. Next, we discuss some of the causes attributing to the differences.

First, the Lilith statistics indicated a greater frequency of loads on local variables than for Mesa. A contributing factor is the use of peephole optimization by the Mesa compiler; thus, "SLWi LLWi" sequences are converted to the equivalent of "COPT SLWi" to save a memory reference. As Figure 3 illustrates, the COPT(copy top of stack) instruction is heavily used by the Mesa compiler. The Mesa software also made much greater use of doubleword variables than the Lilith software(the Modula-2 compiler did not support the LONGINT type). Other differences occurred in the number of procedure calls(more on the Lilith) and in the frequency of use of the "Load Local Address" instruction(more by Mesa). Again, code generation strategies have an effect. The Lilith generates an address cell for every local array or record. As a result, it is difficult to determine whether a "Load Local" is loading an address or a value. The use of an address cell

Opcode Distribution for The Lilith and Alto				
Lilith Mnemonic	Instruction Description	Lilith Count	% of All Instructions	% of All For the Alto
LI	Load Immediate	26534	18.13	16.92
LLV	Load Local Word	22394	15.31	12.68
SLV	Store local variable	6752	4.62	6.57
CDPT	Duplicate top of stack	1823	8.78	5.18
LLD	Load Local Doubleword	412	8.27	5.89
CX	Call external procedure	8318	5.69	4.52
JP, EXC	Unconditional Jump	4852	3.32	4.18
L/SSWB	Dereference pointer on TOS	3551	2.42	3.42
SLD	Store Local Doubleword	358	8.24	3.85
LLA	Load Local variable's Addr	1485	8.96	2.36
UADD	CARDINAL add	3595	2.53	2.34
RET	Procedure return	2688	1.97	1.95
LGW	Load module global variable	9388	6.41	1.74
CL	Call local procedure	4584	3.13	1.73
DADD	Double add	2	8.88	1.57
			65.78%	73.38%

Figure 3

increases the usage of the "short" opcodes while requiring more frame stack space at runtime. Another significant difference arises in the use of global variables by the Lilith's programmers.

3.1 Module and procedure statistics

Figure 4 summarizes the module and procedure statistics that were collected. Most are self explanatory. The "minimum path length" measurements represent our attempt to quantify how long a procedure tends to execute, when it is first invoked, before transferring to another procedure. The idea was to form an estimate of how much time was available to overlap frame initialization and execution. For example, the saving of the return address need not occur before a procedure begins execution as long as it has completed before the procedure exits. The path length is measured in instructions and takes into account all possible execution paths from a procedure's entry point.

Module and Procedure Statistics For The Lilith		
Description	Average	Standard Deviation
Minimum path length to ext. call	19.91	64.18
Minimum path to any proc. call	16.31	76.81
Procedure calls per module	74.83	185.48
Procedure calls per procedure	6.13	35.68
Local calls per module	29.96	57.68
Local calls per procedure	2.45	18.48
Procedures per module	12.21	13.63
Imported modules per module	5.29	3.63
Instructions per module	611.44	971.64
Instructions per procedure	65.45	355.63
Data size per module (words)	35.72	49.52
Words reserved by ENTR/procedure	3.81	17.36

Figure 4

The data sizes listed include the number of words per module and the number of words

allocated on the stack frame on procedure entry. The stack frame word count includes only one or two cells per variable; thus for arrays and records, the address cell was counted but not the space for the data structure. The count does include parameters, however. On the Lilith, arguments are passed by pushing their address or value on the evaluation stack. In the body of a called procedure, then, the arguments are copied into local parameter cells. As a result, both parameters and local variables are addressed by using a positive offset from the L register.

4. Lilith.mmp, Directions For The Future

The Lilith.mmp project involves a set of experiments that are intended to guide the design of a tightly-coupled, multiple-processor, high-level language workstation. In this paper, we are concerned only with instruction set design. The goals of Lilith.mmp are as follows:

Goals of Lilith.mmp

1. Integrated support for lightweight processes.
2. Graphics performance as good as the Lilith.
3. Construction of a Balanced Instruction Set Computer.
4. High-speed context switch.
5. Multiple instruction set processors.

Lilith.mmp is intended for applications that have a requirement for both large numbers of processes and a fast context switch time. As a result, we must find a feasible alternative to the register window schemes that are currently in vogue. The reason is that saving hundreds of window registers on a context switch is slow. A lightweight process is characterized by a small context block and is typically created using Modula-2's InitCoroutine primitive. A heavyweight process would be created using the equivalent of the UNIX "fork" or "exec" system calls.

Also, while we ascribe to the belief that it is advantageous to use pipelining in an attempt to execute an instruction nearly every machine cycle, we are unwilling to jump on the Reduced Instruction bandwagon. The Lilith is justly famous for its code density, an advantage that we are loath to discard. The final three sections of the paper present some of our preliminary ideas, which must still be tested, with regard to code density, balanced instruction processing, and fast procedure calls.

4.1 Code density

A modern processor must be capable of supporting not only a variety of data types (CHAR, CARDINAL, REAL), but also a number of different widths (16, 32, 64 bits) for each type. Typically, it is most advantageous to orient the ALU and data types toward the most frequent width, say 32 bits, and then to implement loads and stores for the other widths to perform either extension or repetition, as appropriate. That is, a 16-bit load would sign-extend to 32 bits and a 64-bit load would be implemented as two 32-bit loads.

The problem with respect to code density is how to encode all of the different types and widths efficiently. Even though our static analysis indicated a low usage for the REAL type, it cannot be disregarded when allocating opcode space. When a program is executing routines in the MathLib module, it should benefit from the same code density as would be found in a CARDINAL sort procedure. The instruction set designer, then, is faced with a choice between giving both types equal opcode space and playing favorites.

Our solution to this dilemma is to overload opcodes; that is, each type is characterized by a certain number of widths, immediate constants, and operators. The assignment of these opcode values is static and is shared by all types. In our first experiments[5], the type associated with the opcode set was selected by means of a MODE instruction. This instruction can be envisioned as setting a base register that identifies the associated microcode or that selects a functional unit, such as a floating-point coprocessor.

The advantage of the MODE approach is that it allows a static set of opcodes to be reused for arbitrarily many types. As a result, all types share equally in the code density advantages of the opcode assignment. Furthermore, the MODE approach avoids retrofitting the instruction set over a machine's lifetime as new types are added.

Chong[5], in an extensive analysis, used a Modula(not Modula-2) compiler to generate a modified MCODE that included a MODE instruction. The static analysis showed that 8.5% of all instructions were MODE settings, while the dynamic analysis demonstrated that MODEs made up 13% of all instructions executed.

4.2 Balanced instruction set processing

One of the most cogent principles of the RISC design philosophy is the so-called N+1 principle; that is, beware of adding an additional instruction if its implementation slows the execution of the N existing instructions. Adherence to the N+1 principle leads one to the conclusion that Reduced Instruction Set Computers are an appropriate design goal. More complex instructions are then coded as subroutines. This results in a space penalty as well as a speed penalty. The graphics on the Lilith provide a good example of the difference between BBLT(Bit BLock Transfer) in microcode and as a subroutine.

Subroutines require frame space and concentrate memory references in accesses to local variables and parameters. We propose a middle ground between a subroutine and a RISC instruction; e.g. a subroutine that can use the evaluation stack for storage of its local variables. The RISC unit would then be considered as a primitive execution engine, or E-Unit. It is conceivable that there might be one EU per type; thus, EUs could be mixed and matched for different applications. The current MODE setting would determine the particular EU to be "applied" to the evaluation stack.

A Balanced Instruction Set Computer would then be composed from RISC execution units and firmware subroutines stored in ROM. The reason that we propose multiple RISC execution units is due to the N+1 rule. As we said, there could be a RISC EU per type. Most manufacturers are currently using this approach for floating point, for example. The other advantage of multiple EUs is the ability to have them operating in parallel. It would be advantageous, for instance, to be able to overlap the execution of a graphics operator with the execution of a floating point instruction.

The use of multiple EUs also leads to the notion of on demand context switching. In this strategy, an EU is switched when a process that is different from its owner requests its use. At this point, the EU must complete its current instruction, save its state in its owner's context block, and then give control to the new process. The alternative to on demand context switching is to wait until all EUs can be halted before making a context switch. With this strategy, the context switch time is bounded by the execution time of the longest instruction in any of the EUs.

If BBLT, polynomial evaluation, bit map searching and all the other low-static count instructions are really necessary and useful (that is, they have a high dynamic-count/execution time product), how do we go about implementing ROM-based subroutines that can allocate frame space on the evaluation stack? Basically, we need an instruction that accesses a stack register using an index that is relative to the top of the evaluation stack. The index has to be relative because a firmware subroutine has no way of knowing which registers are available when it starts execution.

Each firmware subroutine allocates its local variables on the evaluation stack on entry. Expression evaluation then uses the top of the register stack as before. The only difficulty is that each subroutine's local variables may be referenced with a number of different offsets as the stack space used for an expression can vary from one statement to another. The compiler can easily keep track of this bookkeeping detail.

Being able to refer to registers in the evaluation stack also has advantages in other areas. First, it is no longer necessary to copy the arguments to local parameter cells for firmware subroutines; they are already in the correct location. As another example, the Modula-2 compiler implements WITH references by storing the generated address in a local variable, which must be reloaded on each reference to the WITH record. With the proposed architecture, the address could be saved on the evaluation stack and then be "brought up" by indexing. The index option can also be used to retrieve common subexpression values that were saved on the stack.

Rather than using a top of stack indexing scheme, another alternative would be to require that the evaluation stack be empty, except for arguments, when a firmware subroutine was invoked; then the standard register numbering scheme could be used. However, this alternative does not support nested subroutine calls. Furthermore, with true random access, all registers must be saved on a context switch because there is no way of knowing which ones are "live". In the first design, the value of the evaluation stack pointer indicates the number of active cells.

4.3 Fast procedure calls

According to our data, the average number of instructions per procedure was 66 and the average number of procedure calls per procedure was 6. We suspect that the dynamic frequency of procedure calls is even higher. In addition, procedure calls are relatively expensive. Thus, it is important to minimize their cost. There have been a number of good ideas that we would like to explore. However, the Lilith architecture does suggest at least one approach that has not been tried before.

On the Lilith, the arguments to a procedure are passed on the evaluation stack. Furthermore, the stack space used by a procedure's frame can be determined at compile time. As a result, all procedure calls within a particular procedure generate the frame descriptors for the called procedures at exactly the same address. Since a frame descriptor is four words, this space could remain allocated for the duration of a procedure's execution. Of the four cells, only the return address word would need to be written. This design could help to reduce memory traffic, an important consideration in a multiprocessor.

5. Summary and Acknowledgements

Obviously, there are many design decisions to consider. Our plan is to implement a hardware simulator first and then bring up the Lilith's operating system on the simulator. Thus, the effects of design decisions on system performance can be measured in a more realistic way than by testing with only user-level programs.

The Lilith.mmp project is a joint research effort with Modula Corporation and is funded by the Virginia Center for Innovative Technology(CIT-INF-027) and the Office of Naval Research and is supported by grants of equipment and software from Burroughs Corporation and Modula Corporation. Mark Wallitsch, now at Bell Labs, collected the Lilith statistics. Professor Wirth and his group at ETH have also provided invaluable assistance and inspiration.

REFERENCES

- [1] Ohran, R.S., Lilith: A Workstation Computer for Modula-2, Ph.D. Diss. ETH No. 7646, (1984).
- [2] Wirth, N., "From Programming Language Design to Computer Construction," Communications of the ACM 28, 2(Feb. 1985) 159-164.
- [3] Cook, R.P. and N. Donde, "An Experiment to Improve Operand Addressing," Symposium on Architectural Support for Programming Languages and Operating Systems, (March 1982) 87-91.
- [4] Cook, R.P. and I. Lee, "A Contextual Analysis of Pascal Programs," Software-Practice and Experience 12, (1982), 195-203.
- [5] Chong, C., The Design and Implementation Of A StarMod Virtual Machine, Ph.D. Thesis, University of Wisconsin, (August 1983).
- [6] Sweet, R.E. and J.G. Sandman Jr., "Empirical Analysis of the Mesa Instruction Set," Symposium on Architectural Support for Programming Languages and Operating Systems, (March 1982) 158-166.
- [7] McCreight, E.M., "The Dragon Computer System, An Early Overview," Proceedings of the NATO Advanced Study Institute on MicroArchitecture of VLSI Computers, Urbino, (July 1984).
- [8] Wirth, N., "A Comparison of Microprocessor Architectures in View of Code Generation by a Compiler", ETH Technical Report 66, (Dec. 1985).

HIGH-PERFORMANCE CPU IMPLEMENTATION OF THE NEBULA INSTRUCTION SET ARCHITECTURE

R. S. Cheng
RCA Missile and Surface Radar Division
Moorestown, NJ 08057

INTRODUCTION

Limitations of 16-bit architectures such as the MIL-STD-1750 ISA initiated the development of Nebula, a 32-bit ISA standard to provide better support for military applications. Since it was proposed as a military standard, it was optimized for Ada programming language support. Nebula's efficient instruction repertoire, variable length data types, powerful task and procedure control, and extensive error checking interrupt and trapping mechanisms allow efficient implementation of a multi-tasking environment with a high degree of control and data security.

A high-performance CPU subsystem works in conjunction with a two-level hierarchical memory subsystem and operates in parallel with an autonomous I/O subsystem that executes concurrent I/O channel programs. A microprogrammable CPU architecture was adopted for design flexibility and ease of modification. High throughput performance was realized through the use of extensive parallelism and special hardware throughout the architecture.

Designed for implementation using 1.25-um CMOS/SOS technology and a target 3-MIP execution rate, the architecture was extensively verified using Register Transfer Level (RTL) simulation. It was then demonstrated via breadboard implementations in which the design was implemented using MSI/LSI technology to provide a cycle-by-cycle, functional equivalent of the projected VLSI implementation.

The VLSI compatibility of this architecture was demonstrated by replacing core CPU functions in the breadboard with VLSI circuit prototypes implemented with 3-um CMOS/SOS technology. These systems were delivered to the Army for evaluation in 1983, serving as functional representations of the final VLSI implementation.

NEBULA REQUIREMENTS

Nebula is an efficient 32-bit ISA, incorporating numerous features that directly support Ada implementation. A 32-bit, byte addressable, virtual address space facilitates multi-tasking and multi-processor configurations. Variable operand sizes (1, 2, 4, and 8 bytes) and data types allow efficient support of Ada data types. Independent specification of opcode and operand types in a variable-length instruction format allows flexible and efficient operand addressing, and minimizes instruction code space.

Efficient arithmetic, logic, and bit-manipulation instructions are provided for data processing, along with special instructions that support task loading and execution, exception handling, and program control. IEEE standard floating point arithmetic and error-handling are also supported.

The independent context-stack and procedure-based control structures allow efficient implementation of Ada-level procedure calls and parameter passing. Special task-control instructions, the context stack structure, and the memory management scheme together provide efficient support for Ada tasking. Numerous interrupt, exception, and trap-handling mechanisms are defined in the ISA to specifically support the exception and trap-handling mechanisms defined in Ada.

DESIGN GOALS AND STRATEGY

High system throughput, design flexibility, high testability and reliability, and compactness (low power, small size, light weight) were the major design goals. The complex requirements of the Nebula ISA and the VLSI design constraints further complicated the design task.

These numerous design goals and objectives had to be prioritized and quantified so that sensible tradeoffs could be made. In addition to establishing a minimum performance requirement level for each design objective, a higher-level performance goal was also set for attainment. The major design strategies employed are discussed below.

Optimization Based on Instruction Mix

The design was optimized based on the Instruction Mix, which identifies the relative frequency of occurrence of all instructions, addressing modes, data types, and various modes of operation. For example, register and literal operands are used most frequently; typically, 7 or less parameters will be passed in procedure calls; data elements are typically 4 bytes or less in size, and so on. These assumptions were used to optimize the design by focusing on the more frequent operations and needs.

Cost-Efficient Parallelism for Speed

Extensive parallelism was introduced at all design levels to achieve the 3-MIP throughput goal. Instruction prefetching and parallel decoding were overlapped with execution. Data processing and address computations were paralleled by separate ALU facilities. Two banks of register cache were used to expedite context stack switching. All hardware operates in parallel under the control of the highly horizontal microprogram structure. Cost-effective hardware features were used throughout the CPU to increase parallelism.

Hardware Redundancy for Reliability/Maintainability

With high reliability and maintainability as a key design objective, extensive redundant hardware and Built-In-Test features were used to support efficient fault detection and isolation in the CPU design. Chip-level master-slave redundancy, level-sensitive scanning techniques, and a Built-In-Test (BIT) processor were employed to provide (1) on-line fault detection without throughput degradation, and (2) off-line fault isolation down to a replaceable VLSI component.

Programmability for Design Flexibility and Ease of Modification

The programmability of the Nebula ISA design was emphasized to provide the flexibility needed to accommodate future enhancements. A microprogrammable architecture was selected primarily for this reason. However, throughput performance was not sacrificed by providing a highly horizontal control structure with minimum decoding overhead. Although a simple, fixed-cycle synchronous interface was used with the high-speed cache to boost system throughput, design provisions allow wait states to be inserted when slower memories need to be employed. By the same token, variable length CPU cycles are supported; microprogrammable cycle lengths accommodate the different critical paths encountered. This also provides design flexibility for future enhancement and technology insertion.

Careful Consideration for VLSI Implementation

Functional and VLSI partitioning played a major role in the design process, enabling a given architecture to be implemented in a given VLSI circuit and packaging technology. The objectives were to minimize the number of unique chip types and the number of chips in the system, thereby minimizing development and production costs while providing the desired system performance. In general, minimizing the required transistor and pin counts when implementing a particular function penalizes performance. Consequently, careful tradeoff decisions were required. Tight encoding in all non-critical paths minimized the I/O pins required at the expense of more encoding/decoding hardware and time. Redundant hardware was sometimes used for "look-ahead" processing to alleviate critical paths. In some cases, redundant hardware was distributed in multiple chips to enhance throughput or reduce I/O pin requirements. Bit-slicing was also used to reduce I/O pin requirements and maximize common chip types.

To allow optimum use of silicon at reasonable development time and cost, a mixture of custom (handcrafted) design and standard cell techniques were employed in developing the chip set. Standard cell technology, supported by a well-defined set of CAD (Computer Aided Design) tools and layout techniques, was selected for design efficiency. Handcrafted design was employed for regular structures to improve speed and density performance. RCA's 1.25-um CMOS/SOS technology was chosen for performance-critical areas, while bulk CMOS technology was selected for other areas to reduce system cost. The packaging scheme consisted of 132-pin, leadless chip carriers on ceramic substrates, mounted on glass epoxy boards.

Global Optimization to Balance Conflicting Goals

Performance goals and requirements, NRE (Non-Recurring Engineering) development cost, and system life-cycle cost were all significant concerns that affected our Nebula design decisions. Additional hardware generally enhances throughput performance; however, it also increases system complexity, cost, size, weight, and

power, and reduces reliability and testability. Built-In-Test circuitry enhances testability and reliability; however, it may reduce system throughput and add hardware cost and system complexity. The various design goals and performance concerns need to be examined simultaneously in tradeoff studies.

NEBULA PROCESSOR SYSTEM ARCHITECTURE

The complete computer system can be divided into 5 subsystems, as shown in Figure 1: namely the CPU, memory, I/O, Built-In-Test (BIT)/Maintenance Interface, and support subsystems. Each subsystem is briefly described, and the CPU subsystem is discussed in greater detail.

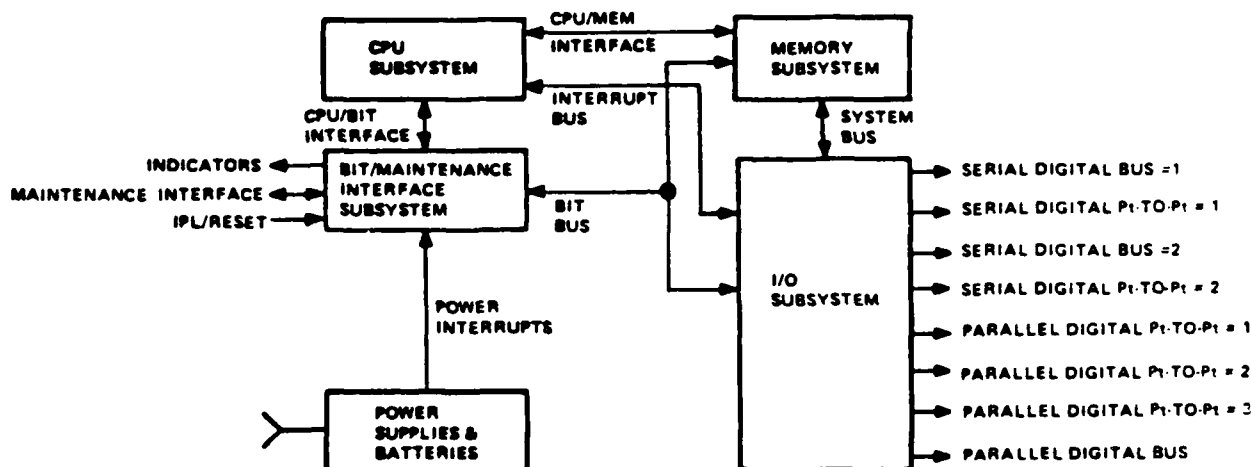


Figure 1. General Block Diagram of AN/UYK-41 Super Minicomputer

The CPU subsystem is a microprogrammable architecture designed to execute the Nebula ISA at a projected 3 MIP execution rate. It contains a highly pipelined control path for instruction prefetching, decoding, and microprogram sequencing functions; 8K x 128-bit microprogram control store for ISA execution and diagnostics; a 32-bit ALU data path augmented with special function units for high-speed data processing; and special interface logic to the other subsystems.

The memory subsystem was implemented as a 2-level hierarchy: a primary storage composed of a high-speed 4K-word set associative cache, and 4 megabytes of bulk memory as secondary storage. Virtual cache was selected to eliminate the address translation from the critical path in order to support fast CPU access. The 4 megabytes of bulk memory were implemented as 2 pairs of 1-megabyte modules interleaved to support efficient cache update. With this configuration, a cache hit will return a 32-bit word in 40 ns from cache. On a cache miss, four 32-bit words are read from bulk memory on two consecutive cycles to update the cache, and the required word is available to CPU in 240 ns.

The I/O subsystem contains independent I/O Processors (IOP) that execute the Nebula instruction set. This subsystem supports high-speed data transfers between bulk memory, and up to seven independent I/O interfaces concurrent to CPU operation. In addition, the I/O subsystem supports data transfers between the bulk memory and the BIT/Maintenance subsystem to support the test and maintenance function.

The BIT/Maintenance Interface subsystem provides the central control for fault isolation and maintenance support. It (1) accepts errors detected by the various BIT circuits distributed throughout the CPU; (2) maintains the fault log, and (3) interrupts the CPU to invoke appropriate diagnostic operations. CPU internal states can be loaded and accessed by a built-in scanning path controlled by the BIT processor, and results can be analyzed for fault isolation.

Finally, the support subsystem is composed of the power supply and the necessary mechanical and thermal support for the complete minicomputer system.

NEBULA PROCESSOR CPU ARCHITECTURE

A highly horizontal, microprogrammable CPU with a microword width of 128 bits was designed to control autonomous hardware units in parallel for high system throughput. Extensive pipelining allows macro-level instruction fetch and decode, as well as microprogram sequencing functions, to be overlapped with instruction execution. The parallelism involved in a typical instruction execution is shown in Fig. 2. Numerous special hardware features efficiently support Nebula-specific operations to enhance system performance. A hierarchical Built-In-Test approach provides concurrent fault detection with no system performance degradation on non-error conditions, and off-line fault isolation down to a specific VLSI component after errors are detected. The CPU operates at a 25-MHz clock rate, allowing a typical register-based instruction to be executed in a single 40-ns microcycle.

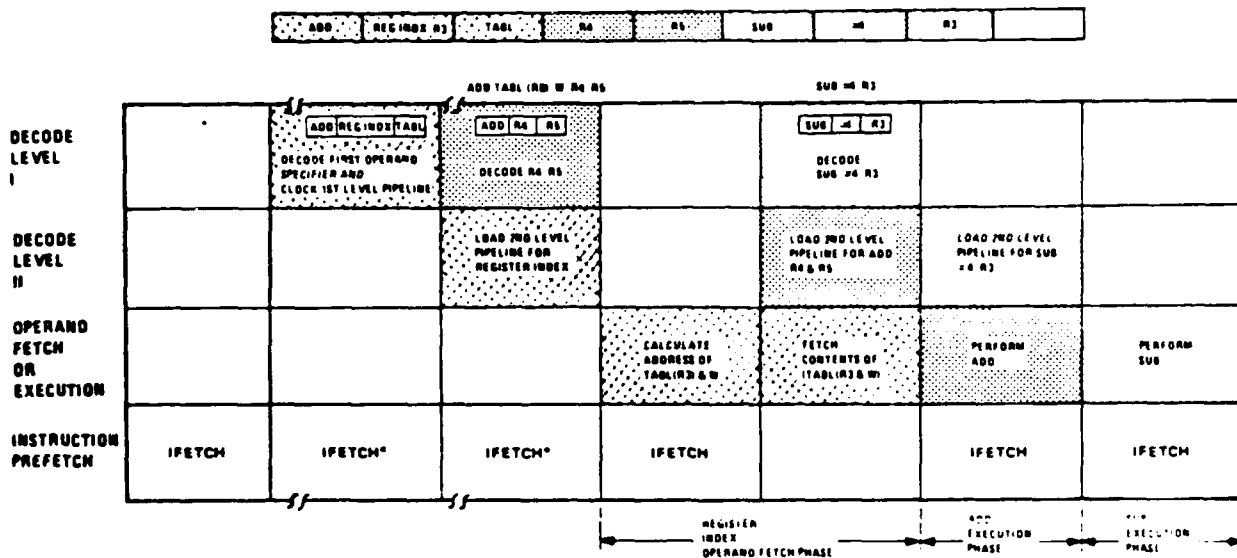


Figure 2. Execution of Typical Nebula Instruction Stream

The complete CPU subsystem is functionally and logically divided into 4 sections, implemented on four ceramic substrates as shown in Fig. 3. These include the Control Pipeline Substrate, the Arithmetic Substrate, the Address Substrate, and the Special Function Substrate. With a total hardware redundancy at the chip level, the CPU subsystem can be implemented with a total of 24 VLSI chips (9 unique chip types) and 8K x 128-bit ROM.

Control Pipeline Substrate (IPU, IDU, MSU)

Three VLSI chip types, the Instruction Prefetch Unit (IPU), the Instruction Decode Unit (IDU), and the Microprogram Sequencer Unit (MSU) are used to implement the CPU control function. A total of 6 VLSI components, 2 of each type, provide a fully redundant control unit, performing Nebula instruction prefetching, decoding, and microprogram sequencing functions. These chips range from 20,000 to 35,000 transistors in size.

One major difficulty posed by the Nebula ISA is the independence between opcode and operand specification in a variable-length instruction format. An instruction typically ranges from 1 byte to over 30 bytes long and, in the special case of a Procedure call instruction, it may be well over 1000 bytes long, supporting up to 256 parameters. A 16-byte instruction queue controlled by a parallel hardware decoding scheme allows effective overlap of instruction fetching and decoding with instruction execution. The selected size of the queue provides the storage needed to support all cases of the prefetching function, yet minimizes the queue refill overhead during branch conditions.

Whenever the queue can accept 4 or more bytes, and the data bus is available, four instruction bytes will be fetched. Instruction data are continually fetched into the queue, and subsequently pipelined into the decoding path. Two hardware pointers generated by the decoding logic extract variable numbers of bytes from the

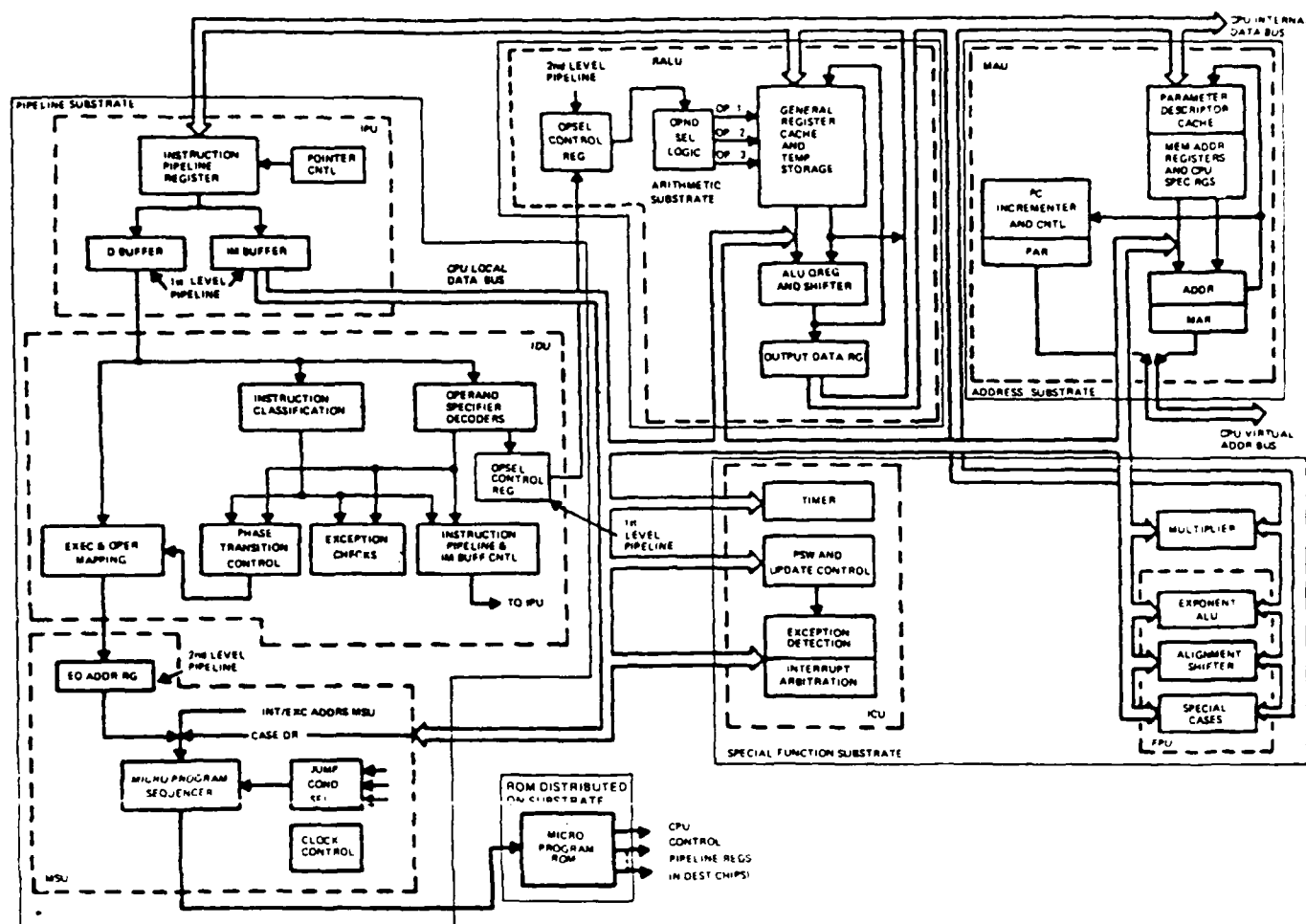


Figure 3. CPU Block Diagram

instruction queue into two levels of pipeline registers. One pointer points to the next opcode or operand specifier to be decoded, while the other pointer identifies the starting byte of the in-line address or data as a result of the last decoding phase.

The in-line data is formatted and then stored into dedicated pipeline registers. Addresses are sign extended and data literals are duplicated to 4 bytes. Simultaneously, register numbers and short literals are encoded to facilitate data manipulation and address calculation in subsequent operand access and execution phases. Error checking is also performed during the decoding process. Any exceptions, including memory protection violations, are pipelined so that the exceptions are raised when execution is attempted.

The opcode/operand specifiers are sent to the decoding path and decoded along with the current opcode and machine status to generate control for the instruction queue and the starting address of the microprogram routines. Since the register and in-line short literal modes are used most frequently, based on the instruction mix, their decoding and set up are completely performed by the decoding hardware and overlapped with current CPU execution. Register numbers or short literals are encoded directly by hardware into operand-select registers, which are used to access the data at execution time. In addition, two consecutive bytes are always decoded simultaneously, allowing as many as two operands to be set up in parallel.

Similarly, for memory operands where address calculation and data fetch are required, the hardware logic will parse the operand specifiers and encode required information before invoking a common set of operand access micro-routines for the operand fetch. After a maximum of 3 operand-access routines, control is transferred to

an opcode-specific execution routine, where operand data are indirectly addressable by the encoded information in operand-select registers. With this approach, all complex addressing modes can be supported, while the simple but frequent register-to-register instruction can still be executed in a single 40-ns cycle. Data size information is also encoded in these operand-select registers, allowing variable data operands to be processed without additional overhead and delay.

A 16-level, Last-In-First-Out (LIFO) stack is included to support micro-level interrupts and subroutine calls, and a 14-bit-wide microprogram address space supports the addressing of up to 16K words of microprogram store. A 16-bit counter and various support logic support multi-way branching, micro-level looping, and flexible conditional case-branches based on externally supplied condition codes. For example, when a short parameter mode is encountered, a 3-bit parameter number and an illegal parameter indicator can be fed from the decoding logic to allow a multi-way branch into unique parameter access or exception routines.

Arithmetic Substrate (RALU)

One VLSI chip type, the Register Arithmetic Logic Unit (RALU), is used to implement the required data processing function. RALU is configured as a 16-bit, bit-slice VLSI device; two RALU devices provide the desired 32-bit internal data path. This substrate is responsible for all the arithmetic and logic operations required to support the instruction set. The RALU is implemented as a 132-pin package containing approximately 45,000 transistors.

To support the frequently used register-addressing mode and a typical 3-operand instruction format, a 36-word by 16-bit customized 3-port register stack was used in the design. Three operand-select registers, directly encoded by the instruction decoding logic (one dedicated for each operand), provide indirect access to the register stack. A fourth operand-select register, loaded via a micro literal, allows efficient indirect access to the stack under microprogram control. The register stack is also directly addressable by microcode in a conventional way, without using any operand-select registers. To expedite switching between the task and kernel context stacks, two register stacks were used. This allows supervisory or interrupt handling functions to be invoked readily on the kernel context stack, without having to save and restore the registers of the interrupted task.

A 32-bit ALU provides the normal arithmetic and logic operations. Special hardware logic is provided to support various Nebula-specific operations. These include sign extension; byte and bit-manipulation; truncation and rounding operations as controlled by a combination of external condition codes; various machine states; and microprogram control.

Address Substrate (MAU)

One VLSI chip type, the Memory Address Unit (MAU), supports all the auxiliary functions such as data operand address computation, parameter decoding/encoding, program counter update, etc., in parallel with data processing on the arithmetic substrate. The MAU is structurally very similar to the RALU, and was implemented as a 16-bit, bit-slice device; two MAUs are used for implementing a 32-bit address computation unit. It contains various special logic to support the auxiliary functions, and operates autonomously in parallel with the arithmetic substrate. The MAU can be implemented as a 132-pin package with approximately 45,000 transistors.

A 32-word by 16-bit register stack caches context-stack information for fast access. The information includes up to seven parameter descriptors and other information associated with the procedure context. Similarly, a dual-register cache approach is employed to expedite kernel-task context-stack switching. Three dedicated address registers specify the memory address for up to 3 operands in an instruction, and such registers are directly accessible under microprogram control.

Special logic is included to encode and decode parameter descriptors to expedite procedure calling (encoding) and parameter access (decoding). The update of the Program Counter (PC), which points to the operand bytes fetched, is performed in the MAU based on the control passed from the instruction decoding logic. In addition to the normal update of the PC, the PC is also pipelined into a backup register at the Nebula instruction boundary. This PC backup register and special logic support program tracing and exception and trap handling, as well as break points.

Special Function Substrate (ICU, FPU, MULT)

The Interrupt Control Unit (ICU), Floating Point Unit (FPU), and a Multiplier (MULT) provide the other CPU auxiliary functions on the special-function substrate. The ICU contains about 45,000 transistors, while FPU and MULT are projected to have 35,000 transistors each.

The ICU provides all the necessary hardware required to support the interrupt and trap mechanism. These include counters to support the four interrupt timers and time-of-day clock; logic to support the maskable software interrupt requests; and a bus interface to accept I/O interrupts from the I/O subsystem. To enhance the interrupt response, a special I/O-interrupt bus and protocol were devised to allow pending interrupts to be polled and prioritized in parallel with CPU execution. The results are then sampled at macro-instruction boundaries, where all interrupts are examined and acknowledged.

In addition to the three preceding classes of interrupts, various run-time exceptions and traps are also handled simultaneously in the ICU, based on a pre-defined priority scheme as specified in the ISA. The Processor Status Word (PSW), which resides in the context-stack memory, is cached in the ICU to facilitate interrupt, exception, and trap checking and handling. Similarly, two PSWs are cached for the Kernel and Task context stacks.

The FPU and MULT were built as hardware-assist units, enhancing the RALU performance on multiplication and floating point instructions. The FPU supports the IEEE standard floating point format as required by Nebula. It was designed to operate in conjunction with the RALU to support the execution of floating-point instructions. The FPU contains barrel shifters, exponent adders, normalization logic, and the extensive error-checking and exception-handling capability required to support the floating point operations. The MULT was designed to accept a pair of 32-bit operands from the RALU and return a 64-bit product, and this provides a 7:1 performance enhancement over the iterative approach supported by the basic RALU capability.

Bus Interface

The various sections communicate over two 32-bit, bi-directional data buses, the Local Data Bus (LBUS) and the CPU Data Bus (DBUS). LBUS supports inter-chip communication within the CPU, and part of the bus can be sourced by microprogram ROM to provide a micro literal for masking and case branching operations. DBUS supports communication between the CPU and the memory subsystem, and also provides additional bandwidth for local inter-chip data transfer.

In-line instruction data are sent from IPU to the various processing units over the LBUS, where they are properly encoded into pipeline registers. Part of the LBUS can be multiple-sourced with hardware condition codes and status by different hardware units in the system. This part of the LBUS and a micro-literal are used to form specific starting addresses in MSU that allow multi-way branches based on hardware status conditions. To preserve microprogram space, a simple alignment and adder logic are included in the MSU, to control the contiguous address space of such micro-subroutines. Special bits in the ICU-based PSW are also routed over LBUS to different parts of the system as they are required for various processing and error-checking operations.

Memory access is accomplished over the DBUS, where memory reads are supported by a 4K-word cache, and a memory write is queued and subsequently written to both cache and bulk memory. The memory is byte addressable as defined by the ISA, while a 32-bit word-aligned memory is physically implemented. One, two, four, or eight bytes of data may be fetched over 1 to 3 memory cycles, depending on the address of the starting byte. Special logic is included in the RALU to efficiently align all memory data during read and write operations. This logic, used in conjunction with the conditional branching capability in MSU, guarantees that all data is accessed in a minimum number of cycles. A memory protection check is performed by a specific memory management unit in the memory subsystem, while exceptions detected will be received by the CPU as a condition code to initiate a memory trap.

Built-In-Test Support (EDC)

Hardware redundancy at the chip level, level-sensitive scanning of internal machine states, and a BIT microprocessor provide on-line concurrent fault detection and off-line fault isolation down to a single chip for replacement. One Error Detection and Correction (EDC) chip is employed on every substrate to collect and report errors to the BIT subsystem, and to interface the CPU with the BIT, allowing CPU diagnostics to run under the control of the BIT processor.

Every VLSI chip in the system is configured as a master-slave redundant pair; both members receive the same system inputs and produce the same outputs. Only the outputs produced by the master chip are used to drive the system. These outputs are also sent to the slave counterpart, where all the redundant outputs are compared. A single chip-fault signal is generated by combining all possible faults. All chip faults are collected by the EDC chip resident on that substrate and communicated to the BIT processor. Having received the error reports, the BIT processor can gain control of the complete system by loading specific BIT commands into the EDC to interrupt and halt the CPU. CPU internal states can then be loaded with a specific test pattern using a pre-defined scan path, and specific diagnostic routines can be invoked. Resulting states can then be accessed and analyzed for fault isolation purposes.

To minimize pin outs and the basic CPU cycle time, the microprogram ROMs are distributed onto the individual substrates to provide the necessary control. One level of microinstruction prefetching is supported, allowing the fetch of the microinstruction to be overlapped with its execution. Recognizing that the microprogram ROM is one of the most error-prone and functionally critical areas in the CPU, single-bit error detection and double-bit error correction is concurrently provided with microinstruction execution. Error detection is performed on the microcode as it is executed. If no error is detected, there is no performance penalty. If an error is detected, the execution will be cancelled by inhibiting the result clock, so that no machine states are changed. The CPU will then be suspended for a cycle in order for the microcode to be corrected (if correctable), or re-fetched and re-checked (if not correctable), and execution resumes in the following cycle. If the error in the microcode repeats after refetch, the CPU is halted, and control is turned over to the BIT processor.

CONCLUSION

A sophisticated ISA like Nebula provides a software programmer with numerous flexible features that are convenient and powerful to use. However, the complexity, reliability, and cost of such implementations are of primary concern.

The system architecture described in this paper provides a 3 MIP implementation of the Nebula ISA using RCA's 1.25 micron CMOS/SOS technology. The architecture concepts and the test approach described are also generally applicable for high-speed processor designs.

Session 3: Custom Chips

Chairperson: Allen J. Smith
University of California at Berkeley

The Arithmetic Cube
and
It's Associated Algorithms

Mary Jane Irwin
Supercomputing Research Center
4380 Forbes Blvd.
Lanham, MD 20706

Robert Michael Owens *
Department of Computer Science
Pennsylvania State University
University Park, PA 16802

March 1986

This work has been supported in part by the Army Research Office under Contract DAAG29-83-K-0126.

Introduction

Digit serial data transmission can be used to an advantage in the design of special purpose processors where communication issues dominate and where digit pipelining can be used to maintain high data rates [DeR]. VLSI signal processing applications are one such problem domain. We have developed a family of VLSI components which have digit serial transmission and which can be pipelined at the digit level. These components can be used to construct VLSI processors which are especially suited to signal processing applications. One particularly attractive such processor is a structure we call the *arithmetic cube* [Owl]. The arithmetic cube can be programmed to solve linear transformations such as convolutions and *DFTs*, has nearest neighbor interconnects, regular layout, simple control, and a limited number of interconnections. Regular layout and simple control derive naturally from the algorithms on which the processor is based. Long wires are eliminated by the nearest neighbor interconnect. High throughput can be achieved by pipelining the processor at the digit level. The arithmetic cube is programmable in the problem size n ; once implemented for a certain size N , smaller problems can be solved on the same implementation without a loss in performance. In addition, the architecture extends to larger N in a regular and automatic fashion.

Table 1 lists the system conventions to which our VLSI components conform [DeR]. The arithmetic cube contains an arithmetic processor, which is built out of adder and multiplier components, fed by a memory.

Table 1. System Conventions

Convention 1:	communication is base 4 digit serial
Convention 2:	digit pipelined, msd first
Convention 3:	high signal = logical 1 low signal = logical 0
Convention 4:	on-chip data stable during ϕ_2 off-chip data stable during ϕ_1
Convention 5:	numerical format is fixed point fraction
Convention 6:	operand word length is fixed and constant
Convention 7:	multiple precision values are not allowed
Convention 8:	components have fixed and bounded latency
Convention 9:	inputs to components are time aligned
Convention 10:	two broadcast control signals are allowed
Convention 11:	one control signal may accompany the data
Convention 12:	primitive components are fine grain in size
Convention 13:	intercomponent signal wires abut or river route

The data communication format is digit serial, where a digit is represented in base 4 signed-digit format [Atk]. The operand digit set is the maximally redundant symmetric signed-digit set for base 4 ($\{-3, -2, -1, 0, 1, 2, 3\}$) requiring three wires to transmit one operand digit in a two's complement encoded format. All components are

digit pipelined with up to three digit operands input and three digit results output. The data communication flow proceeds in a *right directed* fashion (from most significant digit (msd) to least significant digit (lsd)). Operand digits are input one set per clock cycle. The first result digit is generated and output some number of clock cycles after the input of the first set of operand digits. After the first result digit is output, a result digit is output for each subsequent cycle. This *latency* between the when the first digit of data is input and when the first digit of the output is generated is measured as an integer number of digits. We require that all components have a fixed, known latency. Ideally, this latency should be one although this may not always be possible. Our primitive components can be classified as *fine grain*; that is, each component is limited in size so that the many components making up a processor can fit on a single (or a very few) chip(s). Each component can contain no more logic (or area) than that required for a one digit product cell, two digit adder cells, a shift register, some small local control, and local interconnect.

Ideally for VLSI, processors should be meshes or linear arrays of primitive components, interconnected in a nearest neighbor fashion so that the intercomponent interconnects automatically abut. The arithmetic cube, shown in Figure 1, maintains the nearest neighbor interconnect while retaining flexibility. The cube contains an adder array with N adder components ($A_{i,j}$'s) and a multiplier vector with \sqrt{N} multipliers (P_i 's) which feeds a memory array with \sqrt{N} memories (M_i 's) containing \sqrt{N} words each. Figure 1 shows only the digit wide data paths interconnecting the arithmetic components and the memories.

After discussing the two primitive arithmetic components used in the cube, we will illustrate how the cube can be used to compute \mathbf{Y} when "programmed" with \mathbf{H} , \mathbf{B} , and \mathbf{X} where \mathbf{Y} is defined by either

$$\mathbf{Y} = (\mathbf{H} \odot \mathbf{B} \mathbf{X})$$

or

$$\mathbf{Y} = (\mathbf{H} \odot \mathbf{B} \mathbf{X})^T$$

where \mathbf{B} is a predefined $n_0 \times n_1$ matrix whose elements are either -1, 0, or 1. \mathbf{H} is a predefined $n_0 \times n_2$ matrix whose elements are fixed point numbers, \mathbf{X} is the $n_1 \times n_2$ input matrix, and \mathbf{Y} is the $n_0 \times n_2$ result matrix. We will then show how these operations can be combined in various ways to compute the *DFT*.

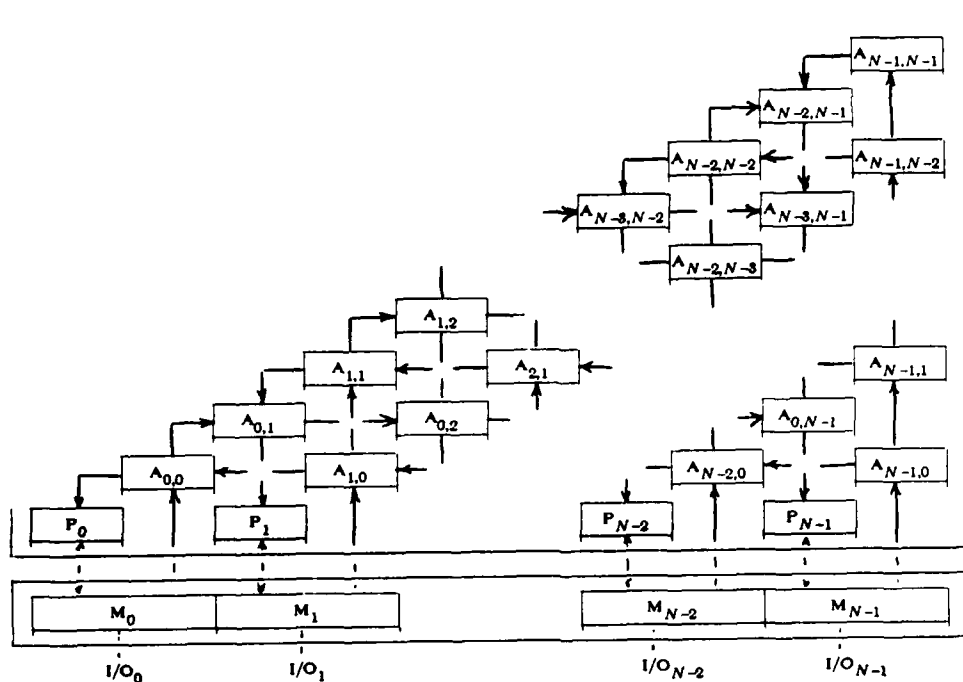
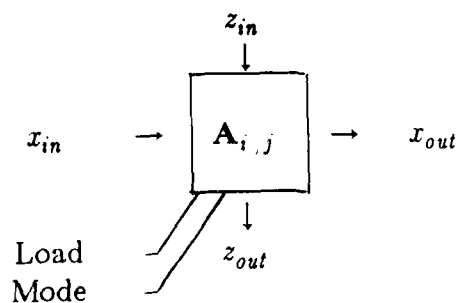


Figure 1. The Arithmetic Cube

Cube Architecture

The arithmetic cube contains an adder array with N programmable adder components ($A_{i,j}$'s), shown in Figure 2, which has been "folded" along the diagonal to maintain nearest neighbor interconnect between the first column of adders and the multipliers. The Load signal loads the p digit circular shift register r_0, r_1, \dots, r_{p-1} . At the end of the Load operation r_{p-1} holds the most significant digit of the loaded operand.



where

Load Cycle (Load = 1):

$$x_{out} = x_{in}$$

$$r_0 = x_{in}$$

$$r_i = r_{i-1} \text{ for } i = 1, 2, \dots, p-1$$

Add Cycle (Load = 0):

$$x_{out} = x_{in}$$

$$r_0 = r_{p-1}$$

$$r_i = r_{i-1} \text{ for } i = 1, 2, \dots, p-1$$

Mode 0 (Mode = 0):

$$(1) s_j = z_{in} + \begin{cases} x_{in} - 4 c_j & \text{if } r_{p-1} > 0 \\ 0 & \text{if } r_{p-1} = 0 \\ -x_{in} - 4 c_j & \text{if } r_{p-1} < 0 \end{cases}$$

$$\text{where } -1 \leq c_j \leq 1 \text{ and } -2 \leq s_j \leq 2$$

$$(2) z_{out} = s_{j-1} + c_j$$

Mode 1 (Mode = 1):

$$(1) s_j = z_{in} + \begin{cases} r_{p-1} - 4 c_j & \text{if } x_{in} > 0 \\ 0 & \text{if } x_{in} = 0 \\ -r_{p-1} - 4 c_j & \text{if } x_{in} < 0 \end{cases}$$

$$\text{where } -1 \leq c_j \leq 1 \text{ and } -2 \leq s_j \leq 2$$

$$(2) z_{out} = s_{j-1} + c_j$$

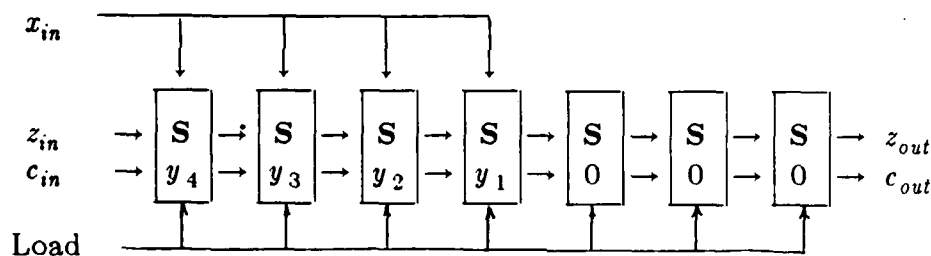
Figure 2. Cube Adder

In the adder, the present sum digit s_j , which is determined by the first digit addition operation (1), is latched for use in the next cycle. Then a second digit addition operation (2) adds the present carry to the previous sum and this result is output. Most importantly this second addition is *guaranteed* to be carry free because of the restricted digit sets allowed for the carry, c_j , and the sum, s_j . The latency of this adder is one digit. The addition component requires two digit adder cells, a one digit storage cell for the present sum digit, and a shift register for R . Only one component is needed regardless of the operand precision. The number of digits of storage provided in the shift register will be the upper limit on the precision of the operands.

Using just the adder array and the memory, the arithmetic cube can be used to compute $Z = B X$. A set of memories is said to contain B (B^T) if the i 'th memory contains the i 'th row (column) of B . With the memories M_i , $0 \leq i < n_1$, initially containing the i 'th column of B , the shift registers in each adder can be loaded by shifting the digits of the elements of B along the data paths connecting the memory and

adder array. Then with the Mode line held at 0, the digits of the elements of \mathbf{X} are cycled through the adder array with the digits of the elements of \mathbf{Z} being collected back into the memories. The time needed is $O(n_0 + n_1 + n_2)$ independent of N . In a similar fashion, the cube can be used to compute $\mathbf{Z} = (\mathbf{B} \mathbf{X})^T$. If the memories initially contain \mathbf{B}^T and \mathbf{X} , \mathbf{Z}^T can be computed by transferring the digits of the elements of \mathbf{X} from the memories to the adder array during the Load cycle to load the shift registers of each adder, and then transferring, while the Mode line is held at 1, the digits of the elements of \mathbf{B} from the memories to the adder array and at the same time transferring the digits of the elements of \mathbf{Z} from the adder array to the memories. The time needed is $O(n_0 + n_1 + n_2)$. Which of the two operations is performed by the adder array is determined by the order in which the operands are supplied and the Mode control line.

The cube also contains a multiplier vector with \sqrt{N} multipliers (\mathbf{P}_i 's). Our semi-systolic, programmable multiplier is built out of \mathbf{S} components each of which contains a digit product cell and two digit adder cells. To multiply two p digit values, $p + 3$ \mathbf{S} components are interconnected as illustrated in Figure 3 where $p = 4$. The \mathbf{S} components are loaded during the Load operation with the multiplier, $Y = (.y_1 y_2 y_3 y_4)_4$, which has been recoded [Atk] so that $y_i \in \{-2, -1, 0, 1, 2\}$. During processing the digits of the multiplicand, msd first, are broadcast on the x_{in} line to the \mathbf{S} components in the multiplier. All other data is passed from component to neighboring component in a systolic fashion. The latency of the multiplier is two.



where for each \mathbf{S}

Load Cycle (Load = 1):

$$z_{out} = z_{in}$$

$$y_i = z_{in}$$

Multiply Cycle (Load = 0):

$$(1) s_j = y x_{in} + z_{in} - 4 c_{out}$$

$$(2) z_{out} = s_j + c_{in}$$

Figure 3. Cube Multiplier

Using just the multiplier vector and the memory, the arithmetic cube can be used

to compute $\mathbf{Y} = \mathbf{H} \oplus \mathbf{Z}$. If the memories \mathbf{M}_i , $0 \leq i < n_0$, initially contain the i 'th rows of \mathbf{Z} and \mathbf{H} , then \mathbf{Y} can be computed by cycling the data values through the multiplier vector. The time needed is $O(n_0 + n_2)$.

If the memories initially contain \mathbf{H} , \mathbf{B}^T , and \mathbf{X} , then

$$(1) \mathbf{Y} = (\mathbf{H} \oplus \mathbf{B} \mathbf{X})$$

can be computed by transferring the elements of \mathbf{B} from the memories to the cube to load the adders and then transferring the elements of \mathbf{H} and \mathbf{X} from the memories to the cube and at the same time transferring the elements of \mathbf{Z} from the cube to the memories. In a similar fashion, if the memories initially contain \mathbf{H}^T , \mathbf{B}^T , and \mathbf{X} then

$$(2) \mathbf{Y} = (\mathbf{H}^T \oplus (\mathbf{B} \mathbf{X})^T) = (\mathbf{H} \oplus \mathbf{B} \mathbf{X})^T$$

can be computed. If $[\mathbf{H}]_{i,j} = [\mathbf{D}]_{i,i}$, $0 \leq i, j < n$, where \mathbf{D} is a diagonal matrix, then

$$(3) \mathbf{Y} = (\mathbf{H} \oplus \mathbf{B} \mathbf{X}) = (\mathbf{D} \mathbf{B} \mathbf{X})$$

is a special case of (1). If $[\mathbf{H}]_{i,i} = 1$, $0 \leq i < n$, and 0 otherwise, then

$$(4) \mathbf{Y} = (\mathbf{H}^T \oplus (\mathbf{B} \mathbf{X})^T) = (\mathbf{B} \mathbf{X})^T$$

is a special case of (2). The time needed to compute any of these four operations is $O(n_0 + n_1 + n_2)$ independent of N , the size of the arithmetic cube. Hence, there is no penalty in solving a small problem on a large arithmetic cube.

Cube Algorithms

We will now describe how these four operations can be used to compute the DFT. The discrete Fourier transform $\mathbf{y} = [y_0, y_1, \dots, y_{n-1}]^T$ of n points $\mathbf{x} = [x_0, x_1, \dots, x_{n-1}]^T$, $DFT(n)$, is given by $\mathbf{y} = \mathbf{W} \mathbf{x}$, where \mathbf{W} is a $n \times n$ matrix whose elements are defined by $[\mathbf{W}]_{i,j} = \omega^{ij}$, $0 \leq i, j < n$, and ω is the n 'th root of unity. The discrete Fourier transform can be computed using *small n algorithms* and their formulations by Good and Winograd [AgC, CoT, ElR, Goo, KoP, Wi1, Wi2]. The original motivation behind the small n algorithms was to develop algorithms to compute either the Fourier transform or cyclic convolution which use as few multiplications as possible. This reduction in the number of required multiplications is at the expense of apparently unstructured sets of additions and subtractions. Since multipliers are usually much larger than adders, it is desirable to minimize the number of multipliers even at the expense of possibly increasing the number of adders. Fortunately, the apparently unstructured sets of additions and subtractions do have a structure which maps in a straightforward manner onto the cube [Owl]. These algorithms are based on the Chinese Remainder Theorem for polynomials and, when data is real, do not use sines, cosines, or complex arithmetic.

The small n algorithms consist of three steps: a set of input additions/subtractions; a set of scalings; and a set of output additions/subtractions represented by three matrices S , C , and T such that $y = S C T x$ where T is a predefined $\delta \times n$ matrix whose elements are either 0, 1, or -1 representing the input additions/subtractions, C is a predefined $\delta \times \delta$ diagonal matrix representing the δ multiplications, and S is an predefined $n \times \delta$ matrix whose elements are either 0, 1, or -1 representing the output additions/subtractions. Optimal small n algorithms have been derived for $n = 2, 3, 4, 5, 7, 8, 9, 16$.

Small n algorithms become impractical for large values of n . For larger n the $DFT(n)$ can be computed by combining an appropriate set of small n algorithms. Three reductions for computing the $DFT(n)$ will be considered. In the following discussion $(S^{n_i}, C^{n_i}, T^{n_i})$ represent the small n algorithm for $DFT(n_i)$, $0 \leq i < l$ where $n = n_0 n_1 \cdots n_{l-1}$ such that (usually) the n_i 's are relatively prime.

A) *The Good's (prime factor) algorithm* [Goo]:

$$Y = \left(S^{n_0} C^{n_0} T^{n_0} \left(S^{n_1} C^{n_1} T^{n_1} X \right)^T \right)^T$$

where X and Y are $n_0 \times n_1$ matrices holding the input and the output vectors. Both n_0 and n_1 must be relatively prime. Good's algorithm can be reformulated as

$$Y = \left(B^1 \left(D^2 B^2 \left(B^3 \left(D^4 B^4 X \right) \right)^T \right) \right)^T$$

where

$$B^1 = S^{n_0}, \quad D^2 = C^{n_0}, \quad B^2 = T^{n_0}, \quad B^3 = S^{n_1}, \quad D^4 = C^{n_1}, \quad B^4 = T^{n_1}.$$

In this form, Good's algorithm uses the series of four cube operations: $3 \rightarrow 4 \rightarrow 3 \rightarrow 4$.

B) *The Winograd algorithm* [Wi1, Wi2]:

$$Y = S^{n_1} \left(S^{n_0} C \otimes T^{n_0} \left(T^{n_1} X \right)^T \right)^T$$

where \otimes is element-wise multiplication and C is the constant $n_0 \times n_1$ matrix

$$[C]_{i,j} = [C^{n_0}]_{i,i} [C^{n_1}]_{j,j}, \quad 0 \leq i < n_0, \quad 0 \leq j < n_1.$$

Both n_0 and n_1 must be relatively prime. Winograd's algorithm can be reformulated as

$$Y = \left(B^1 \left(B^2 \left(H^3 \otimes B^3 \left(B^4 X \right)^T \right) \right)^T \right)^T$$

where

$$B^2 = S^{n_0}, \quad B^3 = T^{n_0}, \quad H^3 = C, \quad B^1 = S^{n_1}, \quad B^4 = T^{n_1}.$$

In this form, Winograd's algorithm uses the series of four cube operations: $4 \rightarrow 1 \rightarrow 4 \rightarrow 3$.

C) *The mixed radix (FFT-like) algorithm:* [OwJ]

$$Y = \left(S^{n_0} C^{n_0} T^{n_0} \left(W \oplus S^{n_1} C^{n_1} T^{n_1} X \right)^T \right)^T$$

where W is the constant $n_0 \times n_1$ matrix

$$[W]_{i,j} = \omega^{ij}, 0 \leq i < n_0, 0 \leq j < n_1.$$

In this case n_0 and n_1 need not be relatively prime. The mixed radix algorithm can be reformulated as

$$Y = \left(B^1 \left(D^2 B^2 \left(H^3 \oplus B^3 \left(D^4 B^4 X \right) \right)^T \right) \right)^T$$

where

$$B^1 = S^1, D^2 = C^1, B^2 = T^1, B^3 = S^2, D^4 = C^2, B^4 = T^2, H^3 = W.$$

In this form, the mixed radix algorithm uses the series of four cube operations:
 $3 \rightarrow 2 \rightarrow 3 \rightarrow 4$.

Conclusions

The decomposition of interest for the $DFT(n)$ is that where a (n_0, n_1) for n exists such that $n_0, n_1 = O(\sqrt{n})$ referred to as an *optimal decomposition*. In this case, the time needed to compute each of the four cube operations reduces to $O(\sqrt{n})$. When an optimal decomposition is used, the time needed to compute $DFT(n)$ is $O(\sqrt{n})$. For example the time needed to compute $DFT(210)$ using the decomposition (14, 15) is $O(29)$ while the time needed using the decomposition (3, 70) is $O(73)$.

For a given precision, each adder and multiplier have constant area. Hence, a VLSI implementation of the arithmetic cube will have area $O(N)$. For an optimal decomposition, the area actually used is $O(n)$ giving an $A T^2$ bounds for the arithmetic cube of $O(n^2)$. For the VLSI models where the time needed to move a given atomic piece of data distance d is bounded by $\Omega(d)$ or where the amount of data which may cross a boundary of length l is bounded by $O(l)$, the $A T^2$ bounds to compute $DFT(n)$ is $\Omega(n^2)$. Within the framework of this model, the arithmetic cube is optimal.

At present the S component was designed in CMOS as part of the MOSIS 1985 summer VLSI course. The cells of the adder are a generalization of a earlier NMOS project. Preliminary results indicate that the arithmetic cube for a reasonably large N (64) will occupy a reasonably small number of chips (16). Such a cube will be able to perform $DFT(n)$ for $n \leq N^2$. The arithmetic cube can also be programmed to compute the cyclic convolution and other linear transformations.

References

- AgC Agarwal, R and J Cooley, "New Algorithms for Digital Convolution," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-25, pp 392-410, 1977
- Atk Atkins, D, "An Introduction to the Role of Redundancy in Computer Arithmetic," *Computer*, 8, No 6, pp 74-76, June 1975
- CoT Cooley, J and J Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Math Computing*, 19, pp 297-301, 1965
- DeR Denyer, P and D Renshaw, *VLSI Signal Processing: A Bit-Serial Approach*. Addison-Wesley, 1985
- EIR Elliot, D and K Rao, *Fast Transforms Algorithms, Analyses, Applications*, Academic Press, 1982
- Goo Good, I, "The Interaction Algorithm and Practical Fourier Analysis," *Journal of the Royal Stat Society*, B-20, pp 361-372, 1958, Addendum, B-22, pp 372-375, 1960
- KoP Kolba, D and I Parks, "A Prime Factor FFT Algorithm using High Speed Convolution," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-25, pp 281-294, 1977
- OwI Owens, RM and MJ Irwin, "The Arithmetic Cube," Department of Computer Science Technical Report CS-85-20, Penn State University, September 1985
- OwJ Owens, RM and J JaJa, "A VLSI Chip for the Winograd/Prime Factor Algorithm to Compute the DFT," to appear in *IEEE Transactions on Acoustics, Speech, and Signal Processing*
- Wi1 Winograd, S, "On Computing the Discrete Fourier Transform," *Math Computing*, 32, pp 175-195, 1978
- Wi2 Winograd, S, "On the Multiplicative Complexity of the Discrete Fourier Transform," *Advances in Math*, 32, pp 83-117, 1979

THE DESIGN OF ULTRA HIGH-SPEED NUMERIC INTENSIVE SYSTEMS (having fault tolerant and load balancing capabilities)

Fred J. Taylor
Dept. of Electrical Engineering
University of Florida
Gainesville, Florida 32611

1. INTRODUCTION

It is axiomatic that to support the design of current and future defense systems, ultra high-speed digital computing machines will be needed. Technologically, there are essentially two ways to build faster digital systems. One way is to use faster electronics and the second is architecture (the art of integrating together functional subsystems). In either case, one normally begins the "top down" design process with a knowledge of whether a general purpose or dedicated machine best satisfies the design goals. University of Florida scholars have a long history of researching these issues through the vehicle of advanced arithmetic unit design, fast algorithm development and integrating these elements together into highly parallel systems in an optimal manner. These individual discoveries have been converging to the integrated design of high-performance, high-speed information and digital signal processing systems for use in satellite communications, emitter identification, radar systems, plus many others. These applications are characterized by single unit (ALU) with a real-time multiply/accumulate rates ranging upwards from 100 M operations per second. In this paper two new architectures for arithmetic intensive applications are proposed. They are theoretically based on

- modular arithmetic (viz: residue)
- homomorphic arithmetic (viz: logarithmic)

We also realize that speed can not be purchased at the expense of cost, power requirements, or reliability. This is especially true in high-volume applications and/or those cases where limited space/power and field maintenance can be expected. As a result, Florida research has been paying much closer attention to these companion issues and, it is felt, made steady progress in this direction during the past year.

2. TECHNICAL BACKGROUND (RNS)

Recently, in major tutorial paper on the subject of the residue number system or (RNS) [Tay84], it was established that this ancient branch of mathematics has enjoyed a renaissance over the last half-decade motivated by the promise of speed. The RNS is specified by a moduli set of L relatively prime integers, say $P = \{p_1, \dots, p_L\}$, $\text{GCD}(p_i, p_j) = 1$ if $i \neq j$. A number in this system is expressed as a L -tuple of integers where the i th element satisfies $X_i = X \bmod p_i$. In the RNS, the sum-difference and product of two long wordlength integers can be computed as a set of L -concurrent fast short wordlength carry-free tasks.

The RNS has often been suggested as a promising media in which to design very high-performance real-time (100 MHz) special purpose signal and image processing systems. The scenario would read as follows:

- Convert a real signal into low wordlength (typ 6-12 bit) data sets using a "flash A/D converter" running at a 10^6 to 10^8 sample rate.
- Map the A/D converted n-bit samples into a RNS L-tuple using direct table lookup conversions. That is, if $\log_2(p_i) \leq m$, m sufficiently small ($m < 6$ bits), use a $2^m \times m$ -bit ROM to map a single X into $X_i \equiv X \bmod p_i$ as a table lookup task.
- Perform arithmetic, restricted to add, subtract, or multiply, using direct table lookups. That is, send the two m bit operands X_i and Y_i to a $2^{2m} \times m$ bit table which contains the precomputed value of $Z_i \equiv (X_i \phi Y_i) \bmod p_i$, $\phi = +, -, \text{ or } \times$.
- Convert an RNS database back into an integer in order to service division related operations or to output results.

Historically, two routines have been used to convert a RNS L-tuple into an integer. One is known as the Chinese Remainder Theorem (CRT) and the other is the mixed radix conversion (MRC) algorithm. They map a residue L-tuple into an integer by using a nested sum of modular partial products. However, the bane of the RNS has traditionally been the inability to support efficient high-speed residue to decimal conversion or (RDC). Unfortunately, it is fundamentally important operation and is found in magnitude compare, sign detection, and overflow management operations and must be dealt with.

2.0 THEORY OF COMPLEX RNS ARITHMETIC

To a small segment of the RNS community, a new wave has been gaining momentum in the area of the complex RNS. Within the framework of a finite ring, one can consider the roots of $x^2 \equiv -1 \bmod p$. If $x \notin \mathbb{Z}$, then x is said to be a non-quadratic root or imaginary [Her75]. Until recently, complex arithmetic was based on these numbers and emulated the classic rules for complex arithmetic which required 2 real adds per add and 4 real multiplies plus 2 real adds per multiply. This is called the Complex RNS or CRNS. Now consider the use of Gaussian primes as moduli which are of the form $p_i = 4n + 1$. When the congruence $x^2 \equiv -1 \bmod p$ has an integer solution (a quadratic root) such that $j^2 \equiv -1 \bmod p$ and $j \in \mathbb{Z}_p$, then j is a real number. It has been shown that there is an isomorphic mapping of a complex number $z = a + ib$ into two new integers $\phi(a, b)$ and $\phi(a, -b)$ under $\phi(a \pm ib) \rightarrow (a \pm jb) \bmod p$ such that

- i. $(a, b) + (c, d) = ((a + c) \bmod M, (b + d) \bmod M)$
 - ii. $(a, b) * (c, d) = ((ac) \bmod M, (bd) \bmod M)$
- (1)

The importance of the result is that only two real multiplications and no adds are required to do complex multiply! We have also proven that the power operation, such as the pervasive magnitude squared mapping $zz^* = |z|^2$, can be completed in one real multiply versus two multiplies plus an add as is currently required in this system which is called the quadratic RNS or QRNS.

Researchers at the University of Florida have achieved several milestones in this area and they are

- Extensions of the theory of the complex residue number system based on rings of Gaussian integers which include:

- i. that the reported QRNS isomorphism does achieve Winograd's lower bound for multiplication complexity [Tay86i].
- ii. the mathematical machinery to synthesize (i.e., derive other than divine) the QRNS isomorphism as well as higher order isomorphisms (i.e., mappings w.r.t. $p(x) = x^2 + 1$ (QRNS), $x^3 + 1$, ..., $x^k + 1$) [Tay86i].
- Implementation of practical complex RNS units based on "off-the-shelf" hardware [Tay85i]. This is suggested by the implementation breakthrough called the single modulus QRNS [Tay85i]. Based on the primitive quadratic roots of $2^n + 1$, for $n = 2, 4, 8, 16, 32$, we can now design complex arithmetic units which we have theoretically shown (based on a gate level analysis) to be potentially 100% faster than conventional (non-residue) units in essentially the same amount of hardware (Figure 1).

To test this hypothesis, the gate array design of a SM unit was undertaken using the GE C20000 CMOS technology [Tay86vi]. The design consisted of the following on-chip modules:

NEG: Negates modulo p	MUTT: Product modulo p
SUM: Carry lookahead adder	JX: Scale by j
MDL: Sum modulo p	ITWO: Scale by 2^{-1}
MUL: Carry-save unsigned multiplier	ITWOJ: Scale by $(2j)^{-1}$

where JX, ITWO, and ITWOJ tasks are essentially radix-2 binary shift operations. For $n = 4, 8$, and 16, execution latencies were simulated using TEGAS which demonstrated a speed advantage for the CRNS. In particular, $QRNS(min)/CRNS(min) = 2.08$, $QRNS(max)/CRNS(max) = 1.32$ which was found to be in support of the theoretical predictions.

- Study of the DFT using recently developed RNS theory [Tay85ii]. Based on a gate-level design study, a conventional 16-bit radix-4 FFT using fast multipliers and CLAs was found to essentially require the same amount of hardware at the butterfly level. If two multipliers are committed to a design, the latency became $35u$ (u = gate level delay) for the conventional unit versus $21u$ for the SM-QRNS!

3. TECHNICAL BACKGROUND (LNS)

The logarithmic number system has been only recently studied seriously [Swa75, Lee77, Swa83, Tay85ii]. A number in this system is approximated by

$$x = \pm r^{\pm e_x}; e_x = [I \text{ bits}: F \text{ bits}] \quad (2)$$

where arithmetic is performed by manipulating exponents, in particular

$$i. \text{ MULT/DIV } C = AB \text{ or } A/B \quad e_c = e_a \pm e_b \quad (3)$$

$$\begin{aligned} \text{ii. ADD/SUB} \quad C &= A \pm B & e_c &= e_a + \phi(v) \\ & & \phi(v) &= \log_r (1 \pm r^v) \\ \text{iii. SQR/SQRT} \quad C &= A^{1/2} \text{ or } 2 & e_c &= e_a/2 \text{ or } 2e_a \end{aligned}$$

It can readily be seen that multiplicative mapping (i & iii) are trivially implemented. The ADD/SUB operation, denoted $\phi(v)$, is not derived but mechanized as a "fast" table lookup mapping. As a result, the precision of the LNS is historically limited to the address space spanned by v . For a high-speed (7-30 ns) lookup latency, this translated to typically a 12-bit range. Recently, Florida researchers have used various data comparison schemes to extend this to 20 bits using a ISL technology. The 62 Kmil² 390 mW 6-function processor chip, with 6-word stack (PUSH/POP, TOP and NEXT TO TOP), possessed the following attributes (Figure 2):

MULT/DIV	40 ns	Range max	(1.84×10^{19})
ADD/SUB	92 ns	min	(5.42×10^{-20})
SQR/SQRT	20 ns		

Precision 2-12.52

4. INTEGRATION

The processors detailed in the previous section are fast and possess a regular dataflow. As a result, they are likely candidates for use as "systolic primitives." The technical approach to integration to be taken is based on some recent published work of Kung, Moldovan, and Fortes [Kun,82, Mol86,For85]. This work describes a method by which an algorithm, described by a difference equation, can be mapped onto the physical rectangular array. These techniques can be used to map algorithms into systolic or wavefront arrays. The algorithm to array mapping is given in terms of an epimorphism (we do not believe it is an isomorphism) matrix T which operates on a matrix of switching primitives. Fortes, for example, has used this concept to develop a gracefully degraded architecture [For85]. Here, if in an $n \times n$ array the (i,j) processor fails, the i th row or j th column (or both) would be taken "off line." This meant that to achieve continuous array processing, at least $(n-1)$ perfectly functional units were removed to protect the system from an isolated fault. Even though this may appear to be "radical surgery," we feel that the mathematical machinery and the refreshing conceptualization of the problem offered by Fortes and others, points us in a very promising direction.

We have developed a method, using incremental difference equations, of predicting the "slope" of these wavefronts. We also feel that we can change the direction of a wavefront through a dynamic (during real-time) redefinition of T . We call this **wavefront steering**. We submit that through the proper application of this concept we can simultaneously achieve both fault tolerance and load balancing. This is a bold statement when it is recalled that it was previously stated that these are generally orthogonal concepts. However, with the steerable wavefront concept can achieve the following:

- Wavefront fault containment: (Figure 3)
- Static load balancing: (Figure 4)
- Dynamic load balancing: (Figure 5)

1. Move the existing tasks aside.
2. Install the interrupting task.
3. Restore the original tasks.

Load balancing can be achieved using one of the following procedures.

- i. After initiating the interrupting task, use the remaining array space to restart some of the interrupted tasks, or
- ii. Do not drive all of the interrupted task to the boundary. Exist only those required to open up sufficient space to begin execution of the interrupted program.

5. SUMMARY

Our residue and logarithmic arithmetic research at the University of Florida has recently overcome several of the problems which has limited its prior use and acceptance. These new findings are now being extended into new areas of complex arithmetic, VHSIC/VLSI and, integration into a viable, highly-parallel numeric intensive processor designs and applications. We will also plan to develop the mathematical machinery which will allow the developed computational primitives to be interconnected into a rectangular array. This study will provide a common framework in which both **fault tolerant** and **load balanced** arrays can be designed and studied. The proposed research in this area will be a synergism of theory, design, architecture, and integration. The end product of this research will be the capability of designing high-performance communications, control, and image processing systems which are necessary to meet many future mission objectives.

University of Florida
1986

6. REFERENCES

- Mol86 P.I. Moldovan and J.A.B. Fortes, "Partitioning and Mapping Algorithms into Fixed Sized Systolic Arrays," IEEE Trans. Comput., Jan. 1986.
- Tay86i F.J. Taylor, "On the Complex Residue Number System," (submitted to IEEE Trans. ASSP and reported at the 7th Symposium for Computer Arithmetic).
- For85 J.A.B. Fortes and C.S. Raghavendra, "Gracefully Degradable Processor Arrays," IEEE Trans. Comput., Nov. 1985.
- Tay85i F.J. Taylor, "A Single Modulus Complex ALU for Signal Processing," IEEE Trans. ASSP, Oct. 1985.
- Tay85ii F.J. Taylor, "A Hybrid Floating-Point Logarithmic Number System Processor," IEEE Trans. on C&S, Jan. 1985.
- Tay84 F.J. Taylor, "Residue Arithmetic: A Tutorial with Examples," IEEE Computer, pp. 50-62, May 1984.

- Swa83 E.E. Swartzlander et al., "Sign/Logarithm Arithmetic for FFT Implementation," IEEE Trans. on Comput., June 1983.
- Kun82 S-Y Kung et al., "Wavefront Array Processor: Language, Architecture and Applications," IEEE Trans. Comput., C-31, Nov. 1982.
- Lee77 S.C. Lee and A.D. Edgar, "The Focus Number System," IEEE Trans. on Comput., Nov. 1977.
- Swa75 E.E. Swartzlander and A.G. Alexpoulos, "The Signed Logarithm Number System," IEEE Trans. on Comput., Dec. 1975.

Acknowledgements: The RNS research was sponsored by the ARO and the LSI by the ONR/SDI office.

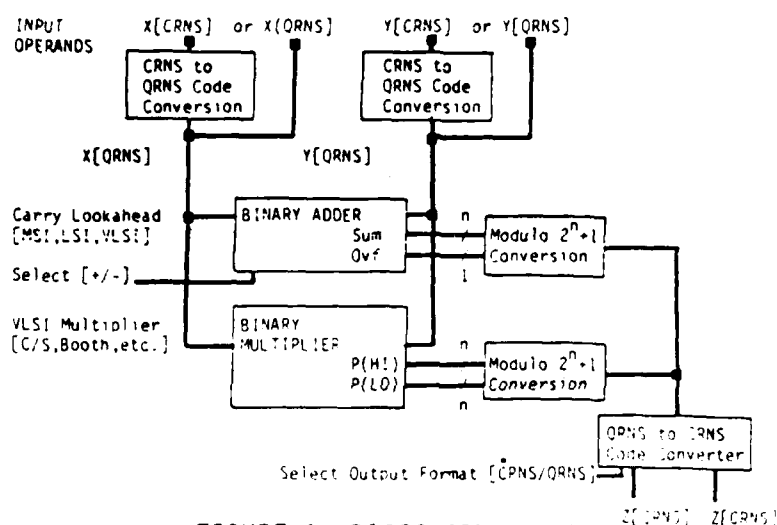


FIGURE 1: BASIC QRNS UNIT

LNS DSP ARCHITECTURE

Functions	MULT	= $x \cdot TOS, TOX \cdot NOS$	40ns
	DIV	= $x / TOS, TOS / NOS$	40ns
	ADD	= $x + TOS, TOS + NOS$	92ns
	SUB	= $x - TOS, TOS - NOS$	92ns
	SQR	= x^2, TOS^2	20ns
	SQRT	= $x^{1/2}, TOS^{1/2}$	20ns
	PUSH X	= shift stack down	
	POP X	= shift stack up	
	X	= external	
	TOS	= top of stack	
	NOS	= next of stack	

Input/Output format

Input = 20-bit LNS word

Output = 21-bit LNS word

Area = 64,300 mil² ; Power = 384 mW

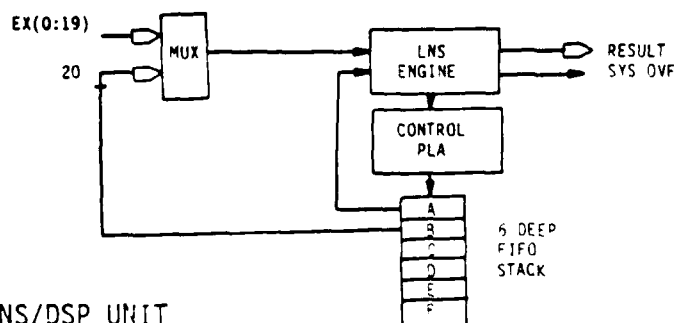
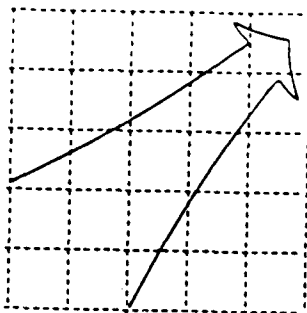


FIGURE 2: SIX FUNCTION LNS/DSP UNIT

UNFAULTED ARRAY



FAULTED ARRAY

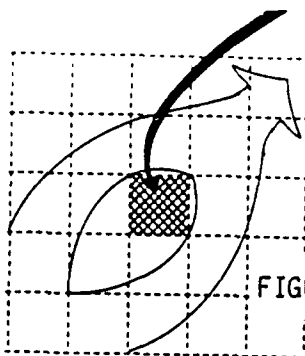


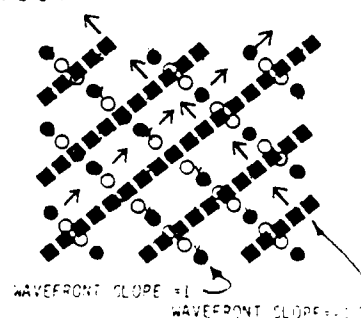
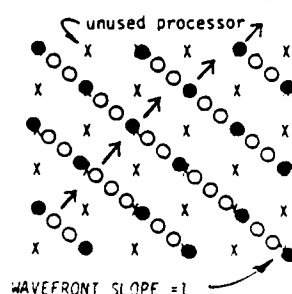
FIGURE 3: WAVEFRONT CONTAINMENT

Optimally Fast Systolic Wavefront

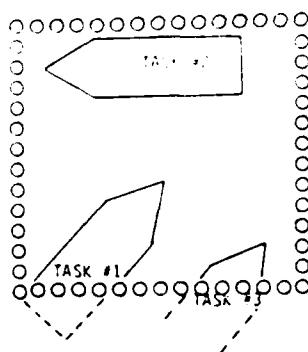
Steered Wavefront

FIGURE 4: STATIC LOADING
AND DATA MANAGEMENT

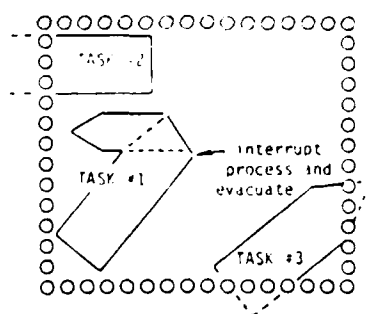
X = RNS PRIMITIVE PROCESSING ELEMENT



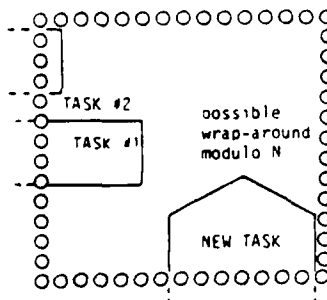
LOAD BALANCING



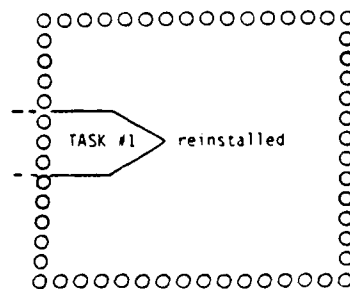
PRIOR TO INTERRUPTS $T=T_0$



INTERRUPT REQUESTED $T=T_1$



INSTALL NEW JOB $T=T_2$



RESTORE UNFINISHED TASK $T=T_3$

FIGURE 5: LOAD
BALANCING

HARDWARE IMPLEMENTATION OF SPECIAL FUNCTIONS
AN APPLICATION FOR COMBAT SIMULATION

by

John B. Gilmer, Jr., Ph.D.
The BDM Corporation
7915 Jones Branch Drive
McLean, Virginia 22102

ABSTRACT

This paper outlines an approach to improving computer performance based on reducing key algorithms to hardware. This takes advantage of very fine grain parallelism which would require either complex control or unacceptable resource management overhead to handle otherwise. As computer aided design and fabrication become more convenient and faster, such reduction to hardware should become as much a programming option as is current use of firmware.

The effort outlined is to explore the means of doing this by reducing a well defined algorithm to a hardware unit that can be plugged into the existing system used to execute the program. The algorithm chosen, hexagonal coordinate system addition, is an important calculation in the CORBAN simulation and similar software of importance to AirLand Battle Management. This algorithm's speedup of two orders of magnitude will allow speedup of the simulation limited only by the fact that its use comprises a fraction of simulation computation. Widespread use of this approach should result in significant speedup when applied to many component algorithms.

A. BACKGROUND

Combat Simulation is currently computationally limited on existing machines. For example, the CORBAN (Corps Battle Analysis) Simulation runs only about as fast as real time on a Sun microcomputer. Orders of magnitude better performance are needed to meet the analytic applications requirements for Battle Management.

There are several approaches to improving performance. One is the improvement of the compiler and other system tools. This may yield a factor of two performance improvement. Another approach is to take advantage of advances in the state of the art. The new 68020 should allow a factor of two or three improvements in run time, albeit at a significant increase in cost. Software structure changes, particularly unpacking of records, can trade space for time to gain perhaps another factor of two. Better algorithms for functions can give speedup as well. For example, a range calculation that currently involves several coordinate transformations can be approximated with acceptable accuracy with arithmetic no more complicated than addition, negation, shifts, and one multiplication.

Parallelism may be profitably applied to make dramatic improvements in run time, but at a high cost in hardware. As the inherent large grain parallelism available is reached, the efficiency of parallelism declines, to the point where marginal costs are not worthwhile. This point is illustrated by previous work that showed, for the Butterfly parallel processor, a decline from 86 percent efficiency to 77 percent efficiency when the number of processors rises from 64 to 124 for a simple simulation with a typical scenario size of 800 units. (1)

Another way of achieving speedup is to migrate algorithms from software to hardware. The effect of using hardware algorithms is to capture fine grain parallelism that can be expressed in a logic diagram but not in serial instructions. This parallelism is unlikely to be captured in any other way, as its fine grain would impose unacceptable management costs or require a very complex machine. Use of a floating point processor in conventional computers is an example of this. But once a fairly limited number of arithmetic and perhaps transcendental functions are so supported with hardware, there remains quite a bit of processing which is not helped by these fairly universal operations. It is on these algorithms that this paper focuses.

The power of special purpose hardware is illustrated by the recent US computer chess championship, where the top two programs, HITECH and BEBE, utilized special hardware which was designed for chess algorithms and were able to beat the previous champion, CRAY BLITZ, which uses a CRAY XMP48. (2)

B. THE CANDIDATE PROBLEM

In the combat simulation CORBAN, there are several utility algorithms that consume a significant portion of the processing. If many of these can be converted into hardware, significant speedups may be achieved in parts of the processing that will not yield much to other approaches described above. One such algorithm has been selected to illustrate this approach and as a candidate for a demonstration.

The CORBAN simulation uses a coordinate system based on a hierarchy of hexagons. (3) (Hexagons are preferred over squares, since in an area so subdivided any two adjacent regions must share a side. This eliminates the corner problem found in square grids. Also, ranges calculated in hexes are accurate enough for most modeling purposes, while this is not true for squares.) Arithmetic performed in the hexagonal coordinate system consumes a significant amount of computation time. As the CPU does not have facilities for such calculations, this must be performed in software.

A utility often called in CORBAN is the Hex addition function, HXADD. On the Butterfly in C, calls to HXADD averaged about four milliseconds each. An assembly language version on a machine faster by a factor of two required about 400 microseconds per call on a 12½ MHz 68000 with no wait states. This improvement is still much slower than the two microseconds or less that would be required to perform the calculation in hardware. If similar speedups can be achieved for a variety of model algorithms, overall performance can be greatly improved, in a manner that can be used in conjunction with other improvements such as parallelism.

C. THE ALGORITHM

The logic of a hardware hex addition algorithm is illustrated in Figure 1. It is of a complexity of about 1000 gates, and requires about one microsecond for gates with 4 ns delay. Table 1 summarizes timing calculations. This is sufficiently quick to make this calculation no longer a factor in the simulation run time. This complexity and performance is well within the capability of what can be fabricated using DARPA's MOSIS facilities. A chip fabricated in this manner can be readily integrated into the system as a memory mapped device, perhaps even inserted as a substitute for memory in one or two RAM sockets.

This particular algorithm is one which is especially suitable for this type of approach for three reasons. First, it has already been put in the fastest form

possible with software, so it is unlikely to benefit from other speedup attempts short of parallelism. Second, it requires no memory references, and thus does not

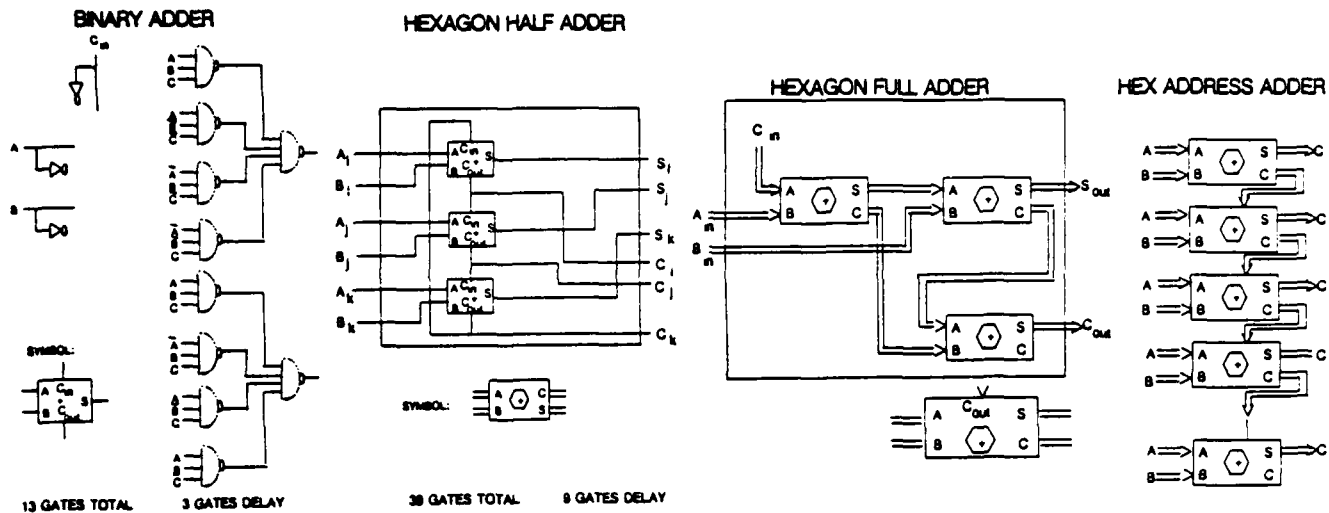


Figure 1. Hex Adder Logic

need to take control of the bus in the manner that a coprocessor would, simplifying the logic. Finally, design and test of such a chip is straightforward because the algorithm is combinational rather than sequential.

TABLE 1. HEX ADDER LOGIC DELAY

	GATES	DELAY
Binary Full Adder	13 Gates	3
Hexagon Half Adder (x3)	39 Gates	9
Hexagon Full Adder (x3)	117 Gates	27
10 Digit Hex Address Adder	1053 Gates	243 = 1 μ s at 4 ns gate delay

Notes:

1. A more optimal design could reduce the delay by adding more gates, reducing the hexagon full or half adders to logic, thus eliminating the intermediate steps.
2. The first adder in the 10 digit hex adder is only a half adder; the last adder does not utilize the carry, so it does not need one of its half adders. Therefore, the total complexity (and delay) is 9 times that of a full adder.

The algorithm is described more fully in Reference 4.

D. EXTENSIONS

Table 2 lists various algorithms in CORBAN which consume significant computation time, about three quarters of the simulation run time, which would benefit from hardware algorithm approach. The several hexagon arithmetic and related functions alone comprize about a quarter of the simulation computation. Other algorithms in the simulation may also have potential for speedup with this approach, but have not yet been investigated. Individually, each one might only give a marginal speedup, but collectively the speedup may be very significant. The hex invert (or negation) operation has a complexity of only about 100 gates, and could be included on the same chip as the hex add. Hex rotation (or multiplication) would also be straightforward, although somewhat more complex than negation. Conversion from hexagonal to (x, y) orthogonal coordinates has a complexity of about 4000 gates, and is quite inefficient in software.

TABLE 2. ALGORITHM COMPUTATION TIME IN CORBAN

Entire simulation, per time step	347.87 sec	
HXADD (Hex addition)	24.16 sec	6.9%
HXINV (Hex negation)	.10 sec	---
HXROT (Hex rotation)	13.62 sec	3.9%
GETHX2 (Hex tree traversal)	66.70 sec	19.2%
PERCEV (Perception, including most calls to HXADD, HXROT, GETHX2, and HXINV)	137.83 sec	39.2%
HAZXYL (Hex to X,Y coordinates)	4.91 sec	1.4%
HXDIST (Hex distances)	10.75 sec	3.1%
AZIMTH (Azimuth angle between hexes)	39.26 sec	11.3%
KILTGT (Target attrition calculation including most calls to HAZXYL, AZIMTH)	64.85 sec	18.6%
MOVSEL (Movement selection, by tree search)	43.55 sec	12.5%
EVALUB (Evaluate decision rules)	7.18 sec	2.1%
Total time of those algorithms listed above	266.93 sec	76.7%
Total time of hex utilities listed above, excluding traversals	92.8 sec	26.6%

NOTE: Totals reflect impact of nested cells.

Another class of algorithms for which a hardware approach would give speed up concerns searching. There are several tree or list data structures which must be searched based on a variety of criteria. For example, one such search traverses a list of units occupying a given hexagon, makes random draws to determine what is perceived, and indicates which are detected. A hardware approach can give about an order of magnitude speedup because all memory references are data accesses necessary for the traversal; none are needed for the overhead of instructions. This type of

device is more complex than the hex addition unit, as it must include addressing logic. It must also be designed to meet coprocessor specifications for the host processor. But this type of algorithm makes up a large proportion of the logic not covered in the previous case, so use of this approach would have a significant benefit. How general in purpose a particular device should be, a matter of cost and risk traded off against utility, will require study.

With some modification to the representation of the searched space, the most often executed portions of the search algorithm could be moved onto a peripheral chip using a small memory array to represent occupancy status for various regions in the model. This would build on the logic of the hex adder. Figure 2 illustrates the logic. This approach captures most of a major model algorithm, currently coded in the "Percev" subroutine, without requiring use of the coprocessor approach.

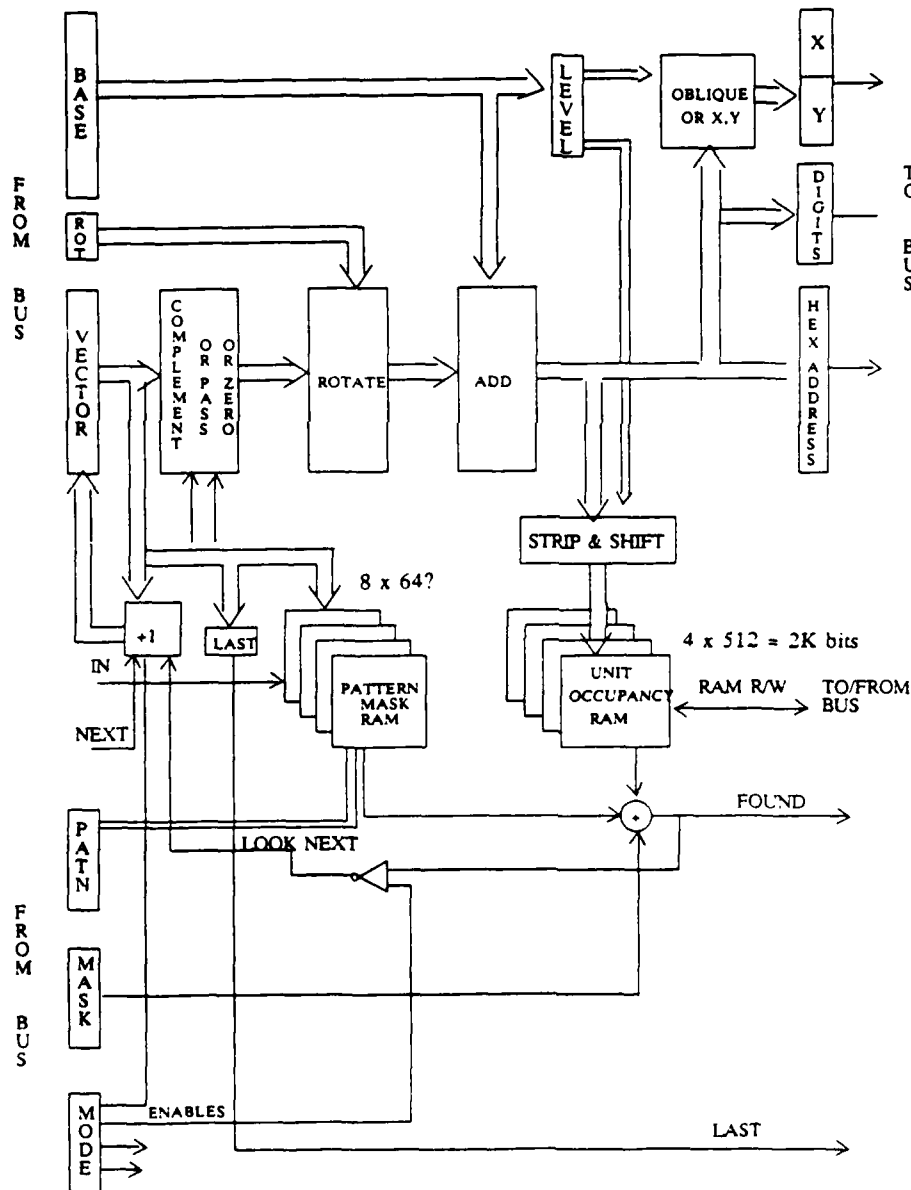


Figure 2. Percev Algorithm Chip

REFERENCES

1. John B. Gilmer, Jr., "Report on the BBN Butterfly Multiprocessor Workshop (August 12-16, 1985): A Battle Simulation Application," The BDM Corporation, September 13, 1985, BDM Technical Report BDM/R-85-0980-TR, page C6.
2. Leon J. Kamin, "Computer Recreations," Scientific American, March 1986, pp. 13-21.
3. John B. Gilmer, Jr., "Dynamic Variable Resolution in the Quickscreen Combat Model," Proceedings of the Winter Simulation Conference, IEEE, December 1984, pp. 597-602.
4. Donald K. Kreckler and Peter J. Lattimore, "An Integrated Coordinated System for Combat Modeling," The BDM Corporation, BDM Technical Report BDM/W-78-297-TR, May 19, 1978.

**Session 4: Memory Hierarchy and Parallel
Architecture**

**Chairperson: C. Forrest Summer
Naval Training System Center**

Research and Development Trends for Memory Hierarchies[†]

Alan Jay Smith
Computer Science Division, EECS Department
University of California
Berkeley, California 94720, USA

Abstract

The effective and efficient use of the memory hierarchy of the computer system is one of the, if not the single most important aspect of computer system design and use. Cache memory performance is often the limiting factor in CPU performance and cache memories also serve to cut the memory traffic in multiprocessor systems. Multiprocessor systems are also requiring advances in cache architecture with respect to cache consistency. Similarly, the study of the best means to share main memory is an important research topic. Disk cache is becoming important for performance in high end computer systems and is now widely available commercially; there are many related research problems. The development of mass storage, especially optical disk, will promote research in effective algorithms for file management and migration. In this paper, we look at each component of the memory hierarchy and address two issues: what are likely directions for development, and what are the interesting research problems.

[†]The material presented here is based on research supported in part by the National Science Foundation under grant DCR-8202501, by the State of California under the MICRO program, by the IBM Corporation, the Signetics Corporation and by the Defense Advanced Research Projects Agency (DoD) under Arms Order No. 4471. Monitored by Naval Electronic Systems Command under Contract No. N00039-84-C-0089.

[‡]This paper is a revised version of a paper presented at the 18th Ann. Hawaii Int. Conf. on Sys. Sciences, January 1986, Honolulu, Hawaii.

1. Introduction

The memory hierarchy of a computer system includes almost all aspects of a computer system's storage, including cache memory, main memory, disk, tape, and mass storage. The effective and efficient use of the memory hierarchy is one of the, if not the most important aspect of computer system design and use. This point is illustrated by comments in two recent papers (see also [Mati84]):

From [Hopk83]:

"The performance of the storage hierarchy may be more important than any details of the computational instruction set. This suggests an approach for those who want to exploit VLSI design: Expend your effort on the memory hierarchy, not on exotic instructions."

From [Mate84]:

"The high performance 16-bit microprocessors introduced over the last five years have broken ground in a new market for microprocessors: high performance systems such as engineering and CAD workstations and even general purpose mainframe level computers. The 16-bit microprocessors generally have plenty of computing power, but suffer in these applications from an inefficient use of memory. The principal purpose of the 16-bit microprocessors now reaching the market is to overcome this difficulty and to provide efficient engines for high performance systems."

"Designing high-performance microprocessor-based systems requires viewing the memory and its buses as the critical elements. DMA, graphics, and multiple CPUs must all contend for this resource, and the key design criterion for CPUs intended for this environment is that they provide high levels of computing power without hogging the bus."

Each component of the memory hierarchy is important for a different reason. The access time to the cache is often the critical path in the CPU. The size and ease of access to main memory affects both performance, through memory access time, paging, swapping and frequency of other I/O, and the ease of programming, in the ability to trade space for running time and programming time. The efficient use of disk affects performance, the size of the disk address space affects the ease of programming, the cost of the disk system is a major component of the cost of the overall computer system, and finally, the physical size of the disks is often a problem in the computer installation. Efficient use of tapes and mass storage impacts system performance, and overall cost by its effect on the number of disks required, operations cost and operations errors.

There are some changes common to all components of the memory hierarchy. First, all memories are becoming logically bigger. This is due to two related trends: memories are becoming denser and are also becoming cheaper, thus it is both physically and economically possible to increase the memory size. Second, the increasing performance of the processor is also necessitating additional memory. The performance of cache and main memories are increasing steadily, as are the capacities of disks.

Among the most important trends in general computer system design is one toward multiple CPU's and distributed systems. This is having and will continue to have an important effect on memory hierarchy design, and as explained below, accounts for some of the most pressing problems in memory hierarchies.

In the remainder of this paper, we present an up to date view of problems, directions and issues in memory hierarchies; this

paper can be considered to be an update of two previous surveys of this topic by the author [Smit78e, 85a]. There will be a section each on cache memories, main memory, gap filler memory, disk, and tapes and mass storage. Some discussion will also be provided on the topic of the "logical" view of memory, as opposed to its performance and physical configuration.

2. Cache Memories

Cache memories are used in modern medium and high speed CPU's and new high end microprocessors to temporarily hold those portions of the contents of main memory which are currently (believed to be) in use. A thorough survey of cache memories appears in [Smit82] and has been updated in [Smit84]; we assume that the reader is familiar with cache memories. An extensive bibliography appears in [Smit86].

2.1. Basic Issues

The two basic performance issues in cache memories are *access time* and *hit ratio*. *Access time* is the time to read or write cache memory, when the desired information is cache resident, and *hit ratio* is the probability of finding the target of the reference in the cache. Access time is crucial because in many, most or almost all computers, *the cache access time is the critical path in the machine* and is the factor most tightly limiting the cycle time and overall machine performance. The hit ratio is important not only for the traditional reason, that it affects the average access time, but also for a relatively new reason: *memory or bus bandwidth is a critical and limiting resource in multiprocessor systems, and the hit ratio directly affects memory traffic*.

2.2. Multiple CPU Systems

Recent trends in computer system technology are encouraging the development of multiple CPU systems. There are two reasons for this tendency: a) The performance of high end CPU's is not keeping pace with the demands for CPU power in certain applications, such as modeling, simulation and numerical computation. b) The cost performance ratio is now significantly better for small machines than large ones, which means that it is cheaper to get a given

"amount" of computation by combining many small machines than having one (or few) large ones.

The basic and critical architecture and hardware issue in multiple CPU systems is that of resource sharing. Such systems, in particular, may share many of the parts of the memory hierarchy, such as main memory, disk and mass storage. There are two resulting problems: (a) maintaining consistency of shared and modifiable information, and (b) avoiding or minimizing queueing and arbitration delays in accessing the shared resources. Cache memories are directly concerned in both problems.

2.2.1. Cache Consistency

In the case that processors have cache memories and also share main memory, the problem is to ensure that the many processors see consistent values of the shared data. There are a number of ways to do that, none of them entirely satisfactory; a more detailed discussion of these appears in [Smit82, 84a, 85e, Swea86]; we summarize here. (1) The cache can be shared; this solution is usually poor, since the cache doesn't have sufficient bandwidth, and the shared design increases the access time. (2) All writes by each CPU can be broadcast to all other CPUs, and the relevant lines either updated or purged; this solution fails for more than 2 or 4 processors, as write traffic begins to interfere with access to each cache. (3) Directory methods (see, e.g. [Arch84] for a recent study) maintain distributed (in each CPU) or centralized (in main memory) directories, that ensure that for each line there is only one CPU able to access a line that has been or is about to be modified. (I.e. many readers or one writer.) Directory methods are expensive to implement and can slow down the caches and memory system due to the need to synchronize use of writable data. (4) In the event that an architecture is new, requirements may be placed on the software, causing it to issue certain hardware commands (e.g. cache purge) that will maintain consistency. This solution is only feasible if the architecture is new, so that old software need not be supported, and if the appropriate synchronizing commands have been implemented. A new way to maintain consistency via software is discussed in [Smit85e], in which "one time identifiers" are

suggested to avoid unnecessary cache purges. (5) A special type of directory method, most suitable for multiple microprocessors sharing the same bus, has recently been developed and is the preferred method for this type of system. (See [Good83], [Swea86].) In it, all microprocessors have logic to watch the bus and ensure that the 1 writer many readers condition holds at all times. The principal limitation of this method is that the bus must be shared; the method does not extend to an arbitrary number of processors. It also needs to be extended to work across bus repeaters.

Each of the above methods has been or is being implemented in one or more systems, but as noted, none effectively and efficiently solves the general problem of maintaining consistency among N processors sharing memory, where N is reasonably large (e.g. >16). *Finding such a solution to the cache consistency problem is a difficult, significant and important research problem.*

2.2.2. Cache and Memory Bandwidth

The second major problem with shared memory is that of memory bandwidth, and this problem is most apparent in the case of multiple microprocessors sharing a single memory bus. (See [Bask76] for an analysis of interleaved memory.) One would like to connect several microprocessors to one memory bus. Using a high performance microprocessor, and a moderate performance bus, it is possible to saturate the bus with from 1 to 5 microprocessors; the addition of more microprocessors brings no increase in overall performance, as the processors spend their time waiting for memory access.

Cache memories are one of the primary mechanisms to solve the memory bandwidth problem. A cache memory can be associated with each processor, either on the microprocessor chip or off-chip. If such a cache has a 16 byte line, a 5% miss ratio per instruction, reads 6 bytes per instruction and writes 2 bytes per instruction, and uses write through, then the memory traffic has dropped from 8 bytes per instruction to 2.8 bytes per instruction. With copy back, and half of the replaced lines being dirty, the memory traffic drops to 1.2 bytes per instruction, an 85% decrease. Thus, cache memories will be necessary for high performance multi-microprocessor computer systems. The trend, then, will be for

future high performance microprocessors to have on-chip or outboard cache memories. (However, it is possible to design a cache memory in such a way that memory traffic actually increases; we are presuming good design.)

2.3. On Chip Cache

With increases in circuit density and chip area, it is becoming possible to place useful cache memories on the microprocessor chip. These caches serve two purposes: they significantly decrease memory access times on hits, and they also reduce the bus traffic. There are a number of design issues regarding on-chip caches; these issues have been addressed primarily in the context of large, mainframe sized caches and need to be reconsidered for small caches. Further, the tradeoffs are somewhat different for on-chip caches, since transfer times dominate latency times for misses, in contrast to larger machines, and main memory traffic is also important. These differences affect optimal choices for parameters such as line size and cache organization; in [Hill84] some of these parameter choices are considered and the sub-block cache organization (sector cache) is analyzed and evaluated. Also, VLSI permits "cheap and easy" associativity, which affects the design (fully associative vs. set associative) of caches and TLBs.

The research issues are ones of evaluating and selecting cache design parameters in the context of small size and limited off chip bandwidth, and the availability of "easy" associativity.

2.4. Off Chip Cache

Of no less interest are off-chip caches. Microprocessors recently announced and/or under development are sufficiently powerful that they can benefit from, and in fact need, more cache than current technology permits to be on-chip. A single cache chip could be expected to hold 4Kbytes or more of cache, as with the Fairchild CLIPPER [Cho86], and a cache board could easily contain upwards of 32Kbytes of cache. Research issues here are the same as apply to cache memories in general, including virtual vs. real address caches, data/instruction caches, multilevel caches, line size, cache size, cache organization and associativity, main memory update algorithm,

and multicache consistency. Some of these are further discussed below.

2.5. Multilevel Cache

There are trends in computer architecture suggesting the further development and use of multilevel cache. On-chip or on-board caches may be too small to be fully effective, but are much faster than more remote caches. Similarly, it may be cost effective to use different consistency methods for each cache level, with different cost and performance tradeoffs. Fujitsu, in its Facom 382 [Fuji82, Hatt83] uses a two level cache and reports that such a design has advantages. We believe that multilevel caches will become more common.

2.6. Virtual Address Caches

Most cache memories are addressed using real addresses; see [Smit82] for a discussion. There are performance advantages, however, for caches addressed using virtual addresses, as is done in the Amdahl 580 [Amda82]. We predict greater use of this design, especially in new designs and architectures where the synonym problem can be eliminated or avoided. (The *synonym problem* occurs when two different virtual addresses map into the same real address.)

2.7. Data and Instruction Caches

Most current cache designs use a single cache which serves for both instructions and data. The advantage to splitting the cache into instruction and data halves is that the bandwidth is doubled; the disadvantage is that if instructions can be modified, then new consistency and correctness problems arise. For newer machine designs, in which compatibility with old, self-modifying software is not a problem, we predict that split caches will become increasingly frequent.

2.8. Microcode Caches

In one special case, that of a cache for microcode, the workload is predictable and static. In that case, the cache can be optimized for the workload, and conversely, the workload (microcode words) can be specially modified to instruct the cache with respect to fetching, replacement and branching. The identification and parameterization of such

optimizations is an open and useful research problem.

2.9. Vector Processors and Caches

Vector processors rely on a steady stream of data to drive the vector unit. Cache memories can be a problem in such a system, since cache misses cause the vector unit to wait until the main memory fetch completes. This can affect performance and can also cause increases in complexity in the vector and cache control logic. (Page faults are an even worse problem.) The proper design of a cache in the context of a vector processor has not, to the author's knowledge, been addressed in the research literature and is an interesting problem.

2.10. Workload and Performance Evaluation

The primary technique used for the performance evaluation of cache memory designs is trace driven simulation. There has been a tendency, however, to use small applications programs for these traces, and then to find that actual cache performance is significantly worse than predicted, since large programs and systems programs tend to dominate. The problem of selecting an appropriate workload is very important to any cache memory evaluation or research effort and is being addressed in [Smit85b].

2.11. Cache Parameter Evaluation

Despite the large number of papers published about cache memories, there is a shortage of hard data in the public literature that the cache designer can use. For example, what is the quantitative effect of varying the line size? What is the quantitative effect of changing the degree of associativity in a set associative memory? The former has been addressed in [Smit85c]. The author is engaged in studies on the latter with a graduate student. The relationship of miss ratio to cache size is considered in [Smit85b, c]. Further quantification of cache design parameters and tradeoffs is needed, especially for caches in a multiprocessor environment.

2.12. And So What Else is New?

Cache memories have existed since the late 1960's and the IBM 360/85 [Lipt68], and

yet are still an active area of research and are an important aspect of computer design; the extensive bibliography in [Smit86] testifies to this. The author also consults widely on the subject of cache memories and with every new design and system, there are special twists that raise new issues. We expect that the activity in cache memory studies will continue for several more years.

3. Main Memory

The use and management of main memory has been an active area of research since the early to mid 1960s; see [Smit78d] for a large bibliography of the relevant literature and [Denn80] for an overview of the research. Traditionally, the research issues have stemmed from the fact that up until the late 1970's, memory was expensive and thus was a critical resource; the efficient use of memory was very important. Problems relating to paging, such as replacement algorithms and control of the degree of multiprogramming were stressed. These problems were exacerbated by the many computer architectures with too few bits of addressing, e.g. 16, which also limited the amount of memory that could be usefully used.

Within the last few years, main memory has become both plentiful and cheap. Thus, except for rare cases, memory is no longer a scarce resource, and the traditional research problems are of much less interest.

There are still some research and development problems relating to main memory. Probably the most important question is *how to design a cost and performance effective shared main memory*. For example, a crossbar, such as was used in C mmp [Wulf72] is expensive, somewhat slow, has some queueing delays [Bask76], and poses reliability problems. For a project at the University of Texas, [Opp83], a banyan network is planned. Many of the various multiprocessor research and development projects have as their central issue the processor and memory interconnection strategy, and its impact on access time, cost, reliability and queueing delays. This is and is likely to continue to be an important and difficult research problem.

Another important question, which has thus far been left almost entirely to the memory manufacturers, is *what is the best*

functional design for a high density memory chip? Should the data come out 1 bit wide, 4 bits wide, 8 bits, or more? If it is desired to read a word at a time, and a word is 32 bits, then 32 chips one bit wide are required; if these are 256Kbit chips, then 32 chips yield a megabyte, which is too much for some applications. It has been remarked to this author that current chip designs are very poor for computer designers; memory chips should be designed to latch the inputs and outputs [Koto84]. Now the external logic to perform this function is expensive and it has a negative impact on performance. Another question is whether "nibble mode" (by which additional sequential bits can be obtained in much less time than the first bit referenced) is useful and represents a good design choice.

The function of *replacement in extremely large main memories* has also been described to the author as a problem. The clock replacement algorithm [Smit78c] is easy to implement given only a reference bit, but the number of page frames that have to be examined becomes unreasonably large in very large memories. The use of set associative replacement for main memories has been previously suggested and analyzed [Smit78a] and was found to be quite effective; the use of set associative clock replacement should provide an effective replacement algorithm yielding adequate performance at low overhead and needing no additional hardware.

Another aspect of the replacement problem is to find a way to have a *consistent reference bit*, when there are multiple cache memories in the system, and the reference bit is maintained locally.

A number of factors suggest that *optimal page sizes will increase* over the next decade. Large pages load and transfer information with fewer page faults, if memory is plentiful. TLBs (of a constant number of entries) can address more memory as page size increases. The number of sets can be increased in real address caches as page size increases, since additional bits are available for addressing without translation. As transmission speeds to and from I/O devices increase, the delay due to large transfers will decrease relative to latency and will become less significant. Right now, 4K pages are a reasonable choice. 512 bytes as is used on some smaller computers is clearly much too small. We believe that

over the next decade, page sizes of 8K or 16K will become desirable.

4. Gap Filler and Disk Cache

The management of main memory has in the past been an interesting research problem because the large gap in access times between main memory (<1 microsecond) and disk storage (10-100ms) meant that transfers, especially due to page faults, often caused substantial CPU idle time. This large access gap still exists, and is likely to get worse, since CPUs and main memory continue to get faster; disk performance has improved very little recently and is not likely to improve overall by a significant amount [Hoag79]. The impact of the access gap still exists, but its focus has shifted more towards explicit I/O. Essentially, the problem is that with the increasing density of disks, their nonincreasing performance and the slowly or nonincreasing number of data paths to disk, *the disk system will be unable to provide sufficient I/O bandwidth to serve high performance CPUs and multiprocessor systems*. This problem is not yet major, but is expected to become worse in the next few years.

The existence of a "gap filler technology", i.e. one intermediate in performance and cost between main memory and disk, would possibly provide a solution to the performance bottleneck projected above. There is, however, no such technology available. Although a few years ago CCDs, magnetic bubbles and EBAM (electron beam accessed memory) were considered promising, none is viable for a gap filler role at this time nor is likely to be soon. (See e.g. [Spec84].)

It is possible to make the useful observation that there is significant locality in I/O reference patterns: data is both referenced sequentially and for some data sets, there is significant reuse, see [Smit85d, Smit75, Smit76, Smit78b] for data supporting this observation. Thus it would make sense to cache portions of the disk address space in main memory or outboard in an MOS RAM based disk cache. The effectiveness of this idea is shown in [Smit85d]. There are already several disk cache products including the Sybercache by SEC, Stors2, and the IBM 4880 model 11, 13 and 23 storage controllers; for both companies' products, the cache is outboard at the storage controller. NEC has a

disk cache which is associated with although outboard to the CPU [Toku80].

The *design of disk caches* has been neglected in the research literature (except for [Smit85d]) and is a fruitful area for study. Questions to be answered include: what is the best location in the system for a cache? How large should the cache be for good performance? What algorithms should be used for fetch and replacement? How large should the blocks be? Should the design be write-through or copy-back, and what are the performance implications of each? The answers to these questions are quite sensitive to the workload and additional workload data needs to be gathered.

The disk caching problem also extends to distributed systems. A recent trend in computer systems is toward a number of processors, including personal computers, workstations, and mainframes linked together with a local area network, and backed by a file server. In such a system, the file server and the network are limited resources, both subject to congestion, and the overheads of remote I/O are substantial. Thus, there are benefits to be gained in caching at the processors, either on local disk or in RAM, and also perhaps to caching in RAM as well at the file server itself, to avoid unnecessary physical disk I/Os. The *research problems of caching in a distributed system* include not only those mentioned above, but the following: should caching occur at the processor, at the file server or both? What policies and parameters are appropriate at each, and are the same or different choices appropriate? How can one maintain consistency in such a system when multiple copies of the same data can exist?

We believe that the use of disk cache will be both more important and more common in the future. The number of interesting research problems in disk cache will also stimulate related research activity.

5. Disks

Disk technology is evolving slowly and has been for some time [Hoag79]. Disk density is increasing at about 20% per year, i.e. it doubles every 3-4 years. Conversely, disk access times are improving very little if at all, with rotation times remaining essentially constant and arm seek times dropping at a slow

rate. These trends should continue at comparable rates over much of the next decade.

It can be expected, however, that *disk I/O transfer rates will increase significantly*. Currently, they reflect the linear density of bits on the disk surface and also the data transmission technology between disk and the CPU. There are three reasons to expect this rate to increase: (a) Increasing physical bit density on the disk surface will increase the rate of reading and writing. (b) Technologies such as optical fibers are becoming available and cost effective as a way to achieve high I/O rates. (c) Performance considerations suggest that higher transfer rates will help alleviate the bottleneck discussed in the section above on gap fillers.

There is a very rapid rate of change and improvement at the "low end" for hard disks. High density small disks are rapidly being incorporated into high end personal computers, workstations and small shared computers. Performance and density will continue to increase and cost will continue drop. Local disks are becoming an important part of the memory hierarchy in a distributed system and will become more so.

Another likely change in disk system design is to *associate more electronics with each disk*. These electronics will act to correct errors, buffer tracks, cache information and/or buffer I/O data streams so as to mask physical latency and decouple transmission from physical disk position and rotation speed. This change is *long overdue* and should begin to occur at any time.

There are a few areas of research in disk systems. *The most promising is the general area of I/O optimization*, which is surveyed in [Smit81b]; see also [Smit81d]. With changes in technology, optimal solutions to issues such as block size, angular placement of blocks, disk loading, etc. change. For a recent implementation of many disk optimizations, see [Mcku84]. Another research problem is to determine the most promising uses of electronics in the disk system.

6. Mass Storage and Tape

We define mass storage to be any storage system with significantly larger capacity than disk, and it is usually cheaper per byte stored and slower than disk. Included in

this category are tape and tape subsystems (such as the IBM 3850, the Ampex Terabit memory and the Calcomp automated tape library) and optical disk. The technology currently undergoing the most rapid development is optical disk. We believe that over the near term, optical storage technology will advance the most rapidly, and within 5 or 10 years, *very large optical archival stores will become common in large computer systems.* No other technology seems to be competitive.

Mass storage can be and is used for explicit I/O, whereby one issues read and write commands to specific devices and data volumes. The most promising use for mass storage, however, is as an automatic backing store for disk. In that circumstance, the disk address space would be expanded, much like virtual memory is used to expand main memory addressing, and automatic migration algorithms would be used to move data between mass storage and disk as needed.

There are numerous research problems relating to the efficient and effective use of mass store as an automatic backing store for disk. Questions such as when to fetch data, how much to fetch, when to remove data on disk and migrate it back to mass store, when to compact mass storage volumes and which volume to transfer it to are all interesting and significant research problems. Some relevant work on file replacement algorithms and file reference patterns appears in [Smit81a] and [Smit81c]; see also [Lawr82] for more work in the same area.

7. Logical View of Memory Hierarchies

In addition to the problems discussed above, which are primarily concerned with the physical design and algorithmic use of memory hierarchies, there are some interesting issues having to do with the logical view of the memory system.

The most important issue is one of *how large an address space is needed.* It has only been recently that there has been a general realization that 16 bits of addressing are not enough, and the newest generation of microprocessors use 32 bits of address. Likewise, the IBM System 370 architecture has been extended in the 308x series machines [IBM82] to use 31 bits of addressing. 31 or 32 bits (4 gigabytes) should be sufficient through

1990 or 1995 for a single processor system, but one can expect to need more addressable memory for a uniprocessor sometime in the 1990's, and earlier for a shared memory multiprocessor system. We expect, therefore, to see architectural changes and extensions to permit this over the next ten years. (Anyone designing a computer at this time would be wise to keep this in mind. S/he should also avoid partitioning the address space via high order bits.)

The idea of *object based architectures* and *capability based addressing* was a popular one in the 1970s, and machines such as the Intel 432, the Cambridge CAP machine and the IBM System/38 all embody and use such concepts. Despite the advantages in programming productivity and data security achieved by that approach, the current belief is that there are inherent performance penalties to such architectures, which suggests instead the use of simple load/store (reduced instruction set -like) architectures with simple addressing. This issue is still an appropriate one for research and it may yet be possible to use a sophisticated logical addressing scheme and yet have good performance.

Protection in most computer systems is primarily associated with memory and the memory hierarchy. Protection bits are usually associated with pages, segments and/or files. Existing protection systems tend to be unable to stand up under sophisticated penetration efforts, and further research in this direction may be warranted.

It has long been a goal of computer designers and users to have a *one level store*, in which all stored information would be addressable within the same address space. Virtual memory is a step in that direction, but does not usually include the file system. We expect some additional research and possibly minor commercial moves in the direction of a one level store, but over the near term, results are not likely to be substantial.

8. Conclusions

In this paper, we have reviewed memory hierarchies, and have addressed two particular points: (a) what are the likely directions for the development of memory hierarchies, and (b) what are the interesting research problems. As was explained in the introduc-

tion, the memory hierarchy is one of the or perhaps the most important part of the computer system with respect to both performance and utility. We therefore believe that with respect to both research and development, memory hierarchies will be a central area of focus over the next decade.

Bibliography

- [Amda82] Amdahl Corp., "580 Technical Introduction", 1982.
- [Arch84] James Archibald and Jean-Loup Baer, "An Economical Solution to the Cache Coherence Problem", Proc. 11'th Annual Symposium on Computer Architecture, June 5-7, 1984, Ann Arbor, Mi., and in SIGARCH Newsletter, 12, 3, June, 1984, pp. 355-362.
- [Bask76] Forest Baskett and Alan Jay Smith, "Interference in Multiprocessor Computer Systems with Interleaved Memory", CACM, 19, 6, June, 1976, pp. 327-334.
- [Cho86] James Cho, Alan Jay Smith and Howard Sachs, "The Memory Architecture and Cache and Memory Management Unit for the Fairchild CLIPPER Processor", February, 1986, submitted for publication. Also available as UC Berkeley CS Report UCB/CSD84/289.
- [Denn80] Peter Denning, "Working Sets Past and Present", IEEEETSE, SE-6, 1, 1980, p. 64-84.
- [Fuji82] Fujitsu Corp., "FACOM M-382", third edition, September, 1982, Tokyo, Japan.
- [Good83] James Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic", Proc. 10'th Ann. Symp. on Computer Arch., June, 1983, pp. 124-131.
- [Hati83] Akira Hattori, Minoru Koshino and Shigemi Kamimoto, "Three Level Hierarchical Storage System for Facom M-380/382",
- [Hill84] Mark Hill and Alan Jay Smith, "Experimental Evaluation of On-Chip Microprocessor Cache Memories", Proc. 11'th Annual Symposium on Computer Architecture, June, 1984, Ann Arbor, Michigan, pp. 158-166.
- [Hoag79] A. S. Hoagland, "Storage Technology: Capabilities and Limitations", Computer, 12, 5, May, 1979, pp. 12-18.
- [Hopk83] M. E. Hopkins, "Compiling High Level Function on Low Level Machines", Proc. IEEE International Conference on Computer Design: VLSI In Computers, November, 1983, Port Chester, New York, pp. 617-619.
- [IBM82] IBM Corp., "IBM 3081 Functional Characteristics", GA22-7076, Poughkeepsie, New York, 1982.
- [Koto84] Alan Kotok, private communication.
- [Lawr82] D. H. Lawrie, J. M. Randal and R. R. Barton, "Experiments With Automatic File Migration", IEEE Computer, July, 1982, pp. 45-55.
- [Lipt88] J. S. Liptay, "Structural Aspects of the System/360 Model 85, II The Cache", IBM Systems J., 7, 1, 1968, pp. 15-21
- [Mate84] Richard Mateosian, "System Considerations in the NS32032 Design", Proc. NCC, 1984, pp. 77-81.
- [Mati84] R. E. Matick and D. T. Ling, "Architecture Implications in the Design of Microprocessors", IBM Systems J., 23, 3, 1984, pp. 264-280.
- [Mcku84] Marshall K. Mckusick, William Joy, Samuel Leffler, and Robert Fabry, "A Fast File System for UNIX", ACM TOCS. 2, 3, August, 1984, pp. 181-197.
- [Oppe83] Eli Oppen, Miroslaw Malek and C. Jack Lipovski, "Resource Allocation in Rectangular CC-Banyans", Proc. 10'th International Symposium on Computer Architecture, June, 1983, Stockholm, Sweden, (also Sigarch News, 11, 3), pp. 178-184.
- [Smit75] Alan Jay Smith, "A Locality Model for Disk Reference Patterns", Proc. IEEE Computer Society Conference, February, 1975, San Francisco, Ca., pp. 109-112.
- [Smit76] Alan Jay Smith, "Analysis of a Locality Model for Disk Reference Patterns", Proc. Second Conference on Information Sciences and Systems, The John Hopkins University, Baltimore, Md., April, 1976, pp. 593-601.
- [Smit78a] Alan Jay Smith, "A Comparative Study of Set Associative Memory Mapping Algorithms and Their Use for Cache and Main Memory", IEEEETSE, SE-4, 2, March, 1978, pp. 121-130.
- [Smit78b] Alan Jay Smith, "On the Effectiveness of Buffered and Multiple Arm Disks", Proc. Fifth Computer Architecture Symposium, April, 1978, Palo Alto, Ca., pp. 242-248.
- [Smit78c] Alan Jay Smith, "Sequentiality and Prefetching in Data Base Systems", IBM Research Report RJ 1743, March 19, 1976, and ACM Transactions on Data Base Systems, 3, 3, September, 1978, pp. 223-247.
- [Smit78d] Alan Jay Smith, "Bibliography on Paging and Related Topics", Operating Systems Review, 12, 4, October, 1978, pp. 39-56.
- [Smit78e] Alan Jay Smith, "Directions for Memory Hierarchies and Their Components: Research and Development", Proc. COMPSAC Conference, Chicago, Ill., November, 1978, pp. 704-709.
- [Smit81a] Alan Jay Smith, "Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms", IEEEETSE, SE-7, 4, July, 1981, pp. 403-417.
- [Smit81b] Alan Jay Smith, "Input/Output Optimization and Disk Architecture: A Survey", Performance Evaluation, 1, 2, 1981, pp. 104-117.
- [Smit81c] Alan Jay Smith, "Long Term File Migration: Development and Evaluation of Algorithms", CACM, 24, 8, August, 1981, pp. 521-532.
- [Smit81d] Alan Jay Smith, "Bibliography on File System and Input/Output Optimization and Related Topics", Operating Systems Review, 15, 4, October, 1981, pp. 39-54.
- [Smit82] Alan Jay Smith, "Cache Memories", Computing Surveys, 14, 3, September, 1982, pp. 473-530.
- [Smit84] Alan Jay Smith, "CPU Cache Memories", to appear in *Handbook for Computer Designers*, ed. Flynn and Roseman

[Smit85a] Alan Jay Smith, "Problems, Directions and Issues in Memory Hierarchies", Proc. 18th Ann. Hawaii Int. Conf. on Sys. Sci., Jan. 2-4, 1985, Honolulu, Hawaii, pp. 468-476.

[Smit85b] Alan Jay Smith, "Cache Evaluation and the Impact of Workload Choice", Report UCB/CSD85/229, Mar., 1985, Proc. 12th Int. Symp. on Comp Arch., June 17-19, 1985, Boston, MA pp. 64-75.

[Smit85c] Alan Jay Smith, "Line (Block) Size Choice for CPU Cache Memories", June, 1985, to appear, IEEE TC. (Also UC Berkeley CS Report UCB/CSD85/239, June, 1985.)

[Smit85d] Alan Jay Smith, "Disk Cache - Miss Ratio Analysis and Design Considerations", ACM Transactions on Computer Systems, 3, 3, August, 1985, pp. 161-203

[Smit85e] Alan Jay Smith, "CPU Cache Consistency with Software Support and Using 'One Time Identifiers'", Proc. Pacific Computer Communications Symposium, Seoul, Republic of Korea, October, 1985, pp. 142-150. Also available as UC Berkeley CS Report UCB/CSD84/290.

[Smit86] Alan Jay Smith, "Bibliography and Readings on CPU Cache Memories", February, 1986. Computer Architecture News, 14, 1, January, 1986, pp. 22-42.

[Spec84] "Whatever Happened to Magnetic Bubble Memories", IEEE Spectrum, September, 1984, p. 22.

[Stor82] Storage Technology Corporation, "Sybercache 8890 Intelligent Disk Controller", Louisville, Colo. 1982. Proc. IFIP 1983, pp. 693-697.

[Swea86] Paul Sweazey and Alan Jay Smith, "A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus", to appear, Proceedings of 13th Int. Symp. on Computer Architecture, Tokyo, Japan, June, 1986. Also available as UC Berkeley CS Report UCB/CSD84/220.

[Toku80] T. Tokunaga, Y. Hirai and S. Yamamoto, "Integrated Disk Cache System With File Adaptive Control, Proc. IEEE Computer Society Conference, September, 1980, pp. 412-416.

[Wulf72] W. Wulf and C. G. Bell, "C.mmp, a Multi-Mini Processor", Proc. FJCC 1972, pp. 766-777.

SIGNAL AND DATA PROCESSING FOR IR SENSORS

D. E. Satterfield
U.S. Army Strategic Defense Command
P.O. Box 1500
Huntsville, Alabama 35807-3801

R. C. Styles
Teledyne Brown Engineering
Cummings Research Park
Huntsville, Alabama 35807

Introduction

Over the past 3 years, the U.S. Army Strategic Defense Command has investigated advanced signal and data processing technology for obtaining information with complex infrared (IR) sensors. The results of these investigations have culminated in the development of a very sophisticated hardware-in-the-loop (HWIL) signal and data processing testbed. This testbed, shown in Figure 1 below, is located at the Army's Advanced Research Center in Huntsville, Alabama. This testbed facility is now being used to conduct research in distributed data processing architectures, algorithms, and real-time applications software for a wide range of missions.

The signal and data processing testbed consists of a System, Environment, and Threat Simulation (SETS) and an Algorithm Testbed Simulation, both of which run on an array of ten VAX 11/780 computers. This nonreal-time code encompasses all algorithm performance enhancements without regard to real-time constraints and serves as a standard by which the real-time algorithms are compared.

These simulations were researched and established by Nichols Research Corporation in Huntsville, Alabama.

The Mosaic Sensor Emulator Unit (MSEU) emulates a sensor module containing 1,920 detectors and outputs real-time waveforms from each detector for 1,000 targets for 20 scans (20,000 targets for 1 scan). The Auxiliary Sensor Signal Processing Unit (ASSPU) performs time-dependent processing [1,700 Million Instructions Per Second (MIPS)], data partitioning processing (556 MIPS), and object-dependent processing (64 MIPS) before presenting three-color information to the data processor. These systems were designed and built by Boeing Aerospace Company in Seattle, Washington. The Advanced Distributed Onboard Processor (ADOP), a 15-MIPS system, then does the measurement processing, color-to-color correlation, scan-to-scan correlation, precision track, discrimination, and threat reporting. This system was designed and built by Honeywell, Inc., in Clearwater, Florida. The real-time application code and part of the operating system were designed and implemented by TRW in Huntsville, Alabama.

Hardware Architecture Descriptions

The MSEU and ASSPU systems are expertly described in the Wednesday, November 6, evening classified session paper entitled "Electronic Emulator and Signal Processor for BMD Mosaic IR Sensor" by Norsworthy and Franzel and will not be repeated here.

The ADOP comprises five processing elements (nodes) that communicate on three, 1-Mword/sec independent global buses. Each node contains three Central Processing Units (CPUs), three Floating Point Processors (FPPs), one million (1M) words of 17-bit memory, and a global Bus Interface Controller (BIC).

The ADOP node architecture, shown in Figure 2, comprises four basic functions:

(1) A fixed point, MIL-STD-1750A, CPU capable of executing the fixed point Digital Avionics Instrumentation System (DAIS) instruction mix at 1 MIPS. The CPU implements a pipelined instruction queue and embedded memory management unit function to achieve this throughput while addressing up to 1M words of memory. The basic CPU microcycle time is 350 nsec.

(2) A FPP serves as an adjunct to the CPU to facilitate achievement of 1 MIPS full DAIS floating point operation. Using the built-in

function generator option of MIL-STD-1750A, special macro instructions can be coded to perform special operations for signal processing operations. The FPP contains a hardware multiplier and an ALU, which essentially "shadows" the CPU arithmetic logic unit.

(3) A four-port local memory organized as 256K words by 17 bits (1 parity bit). A read-modify-write function facilitates semaphore operations required by the operating system. The system of node-local buses provides access to all of memory by each CPU.

(4) A BIC, when coupled with the real-time operating system, efficiently handles all global communications. The BIC accesses message queues in node-local memory and moves messages on the system of Global buses. A logical addressing scheme allows the routing to be configured by the operating system without affecting application software.

The ADOP node structure allows virtually contention-free processing of an application program when data sets are partitioned nonoverlapping. To this end, a complete set of software tools has been developed, including a PASCAL compiler and a distributed operating system. The use of these tools provides a strict adherence to programming standards and configuration controls that make testing on the multi-CPU, multi-node architecture an achievable task.

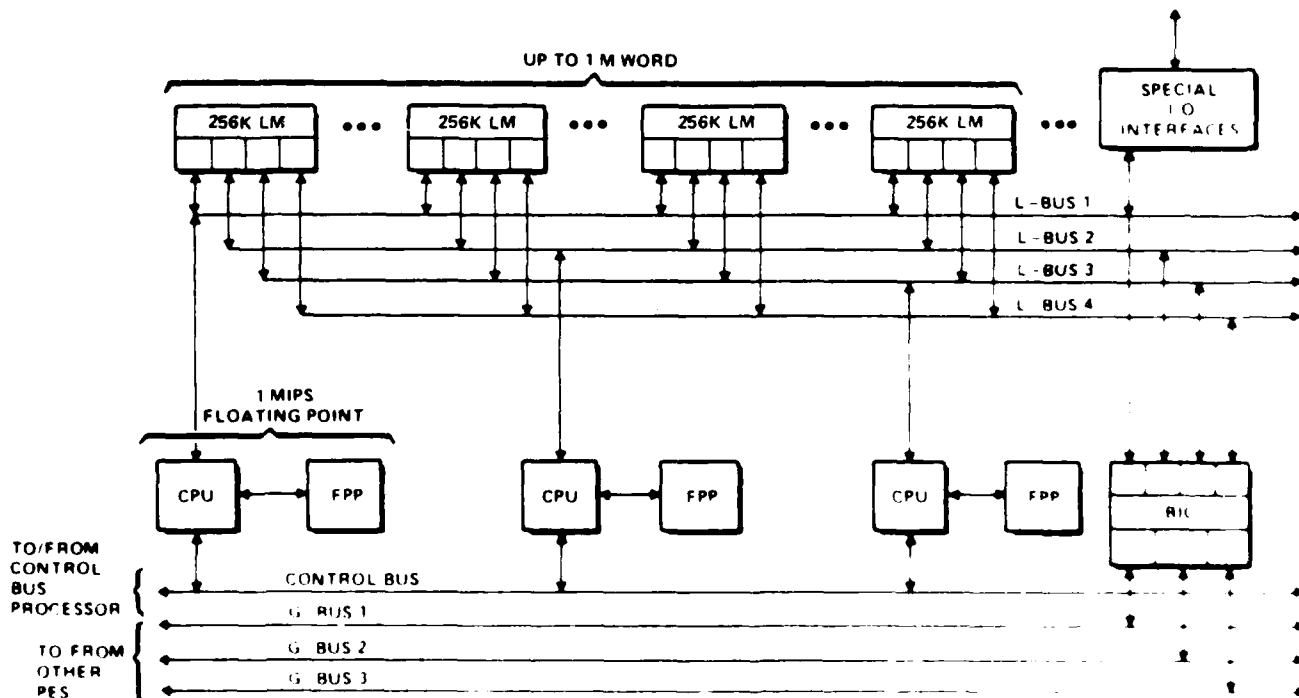


Figure 2. ADOP Node General Data Flow Diagram

Testbed Utilization

The testbed, in whole and in part, provides a facility for several SDI research endeavors, including the Airborne Optical Adjunct (AOA), the Airborne Optical Sensor (AOS), nonnuclear interceptor avionics, and red-blue situations. One of the first research efforts on the testbed was conducted by Teledyne Brown Engineering (TBE), Huntsville, Alabama, using the currently implemented algorithm set based on an optical probe. The TBE research team studied the sensitivity of the software/hardware performance to varying threat characteristics such as the number of reentry vehicles, angular rates, object spacing, and "clumped" objects. These researchers are now studying other proposed algorithms, as well as new reformulations of the existing ones, for improved performance.

Development of threat scenario data is the first step in the utilization of the testbed. At the present time, all of the threat data utilized was developed by TBE as part of the Forward Acquisition System Integrated Ground Test (FAS IGT) Program. These data characterize the threat in terms of the number of reentry vehicles, decoys, fragments, and other objects; the spacings; angular rates; irradiance parameters; etc. The scenario must be designed in such a way that it accomplishes two purposes. First, it must be credible in terms of its representation of a potential ballistic missile attack based on available intelligence data. Second, it must be designed to provide the stressing criteria required to evaluate the hardware/software under analysis. A general scenario as it appears in the Field of View (FOV) of a typical scanning mosaic sensor is shown in Figure 3, with the focal plane array projected into the FOV for clarity. The threat data becomes the "truth" upon which performance measures for the hardware and software are determined. In practice, a subset of the FAS IGT threat that meets the previously defined criteria is selected. This subset is then used as the primary data input for both the SETS and the MSEU.

The threat data, as derived from the FAS IGT scenario, is in the proper form for input to the

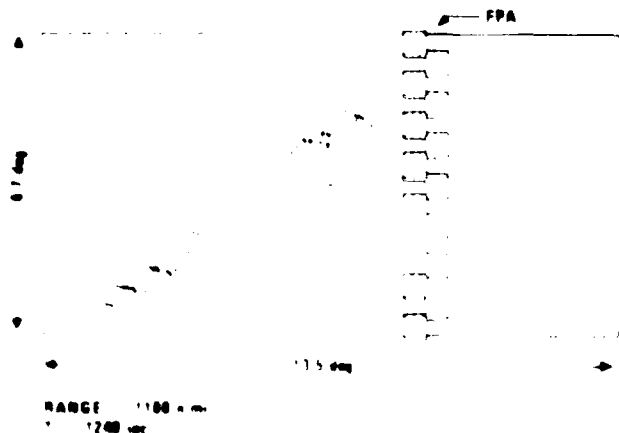


Figure 3. Scanning Mosaic Sensor Threat.

SETS; however, before it can be used to drive the MSEU, it must be converted into instructions that the emulator can use for generation of the appropriate sensors signals.

A series of sophisticated software programs are used to transform the basic threat data into specific optical image data for each detector in the sensor focal plane array. These data form the basis for creation of the MSEU instruction set. The MSEU, in turn, generates the signals that are used as input to the ASSPU. These signals include detector, background, and gamma noise to further stress the ASSPU. In both the SETS and the MSEU-ASSPU, the threat data is processed so that the output from each is in a common format and forms the input data for either the data processing algorithm simulation, the reformulated algorithms, or the ADOP.

The algorithm simulation contains a full complement of maximum performance algorithms that run in nonreal time on the VAX 11/780 network. The output from this simulation is then compared to the "truth" data to establish a new standard to compare with the real-time algorithms or actual performance. The output of the SETS or signal processor is next used to drive the reformulated algorithm set, again in nonreal time. This reformulated algorithm set is the set of algorithms that have been designed (reformulated) from the maximum performance algorithms to run in real time when mapped onto the ADOP hardware. The results derived from the reformulated runs are used to predict the performance of these algorithms when implemented on the ADOP.

The final step is to use the output from the SETS and/or the signal processor to exercise the real-time algorithms on the ADOP to determine the actual performance. This testbed configuration, when used in the manner just described, can support several important research projects for the Strategic Defense Initiative (SDI) Program. A typical output from the system is depicted in Figure 4, which shows the "truth" performance, the algorithm simulation performance, and the ADOP performance.

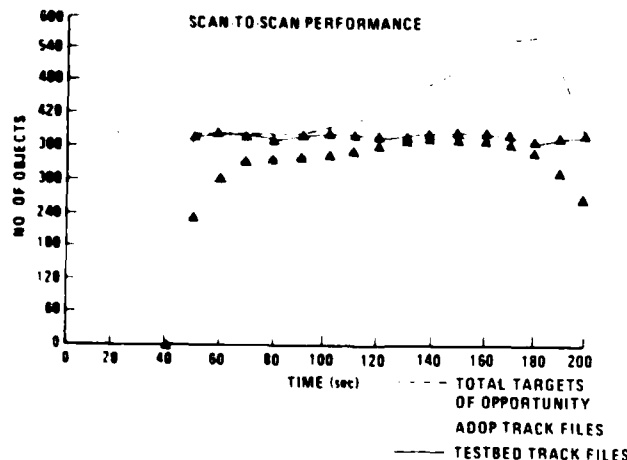


Figure 4. Scan-To-Scan Performance.

Summary

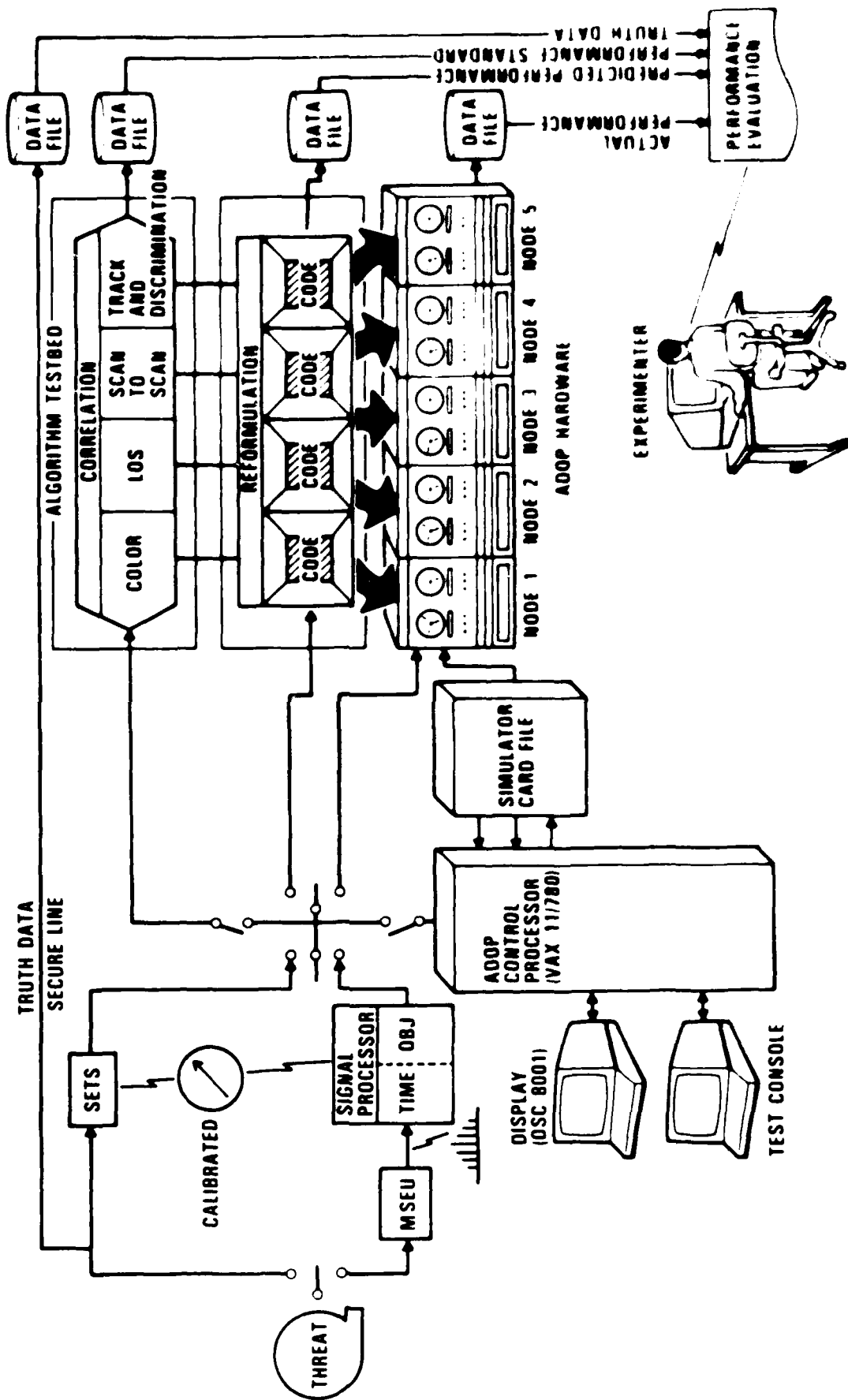
The Data Processing Directorate, within the U.S. Army Strategic Defense Command, working in concert with Teledyne Brown Engineering, Honeywell, Boeing, TRW, and NRC has developed an advanced long wave infrared (LWIR) signal and data processing testbed for studying and characterizing performance for various BMD missions. A full set

of LWIR sensor signal and data processing algorithms has been developed for a probe-type sensor and many experimental results have been obtained. In addition, a complete complement of support software has been developed for this HWIL facility. Analysts are currently conducting research studies on several key technology issues for the SDI Program.

UNCLASSIFIED

HARDWARE-IN-THE-LOOP SIMULATION (U)

ACB 060128 01

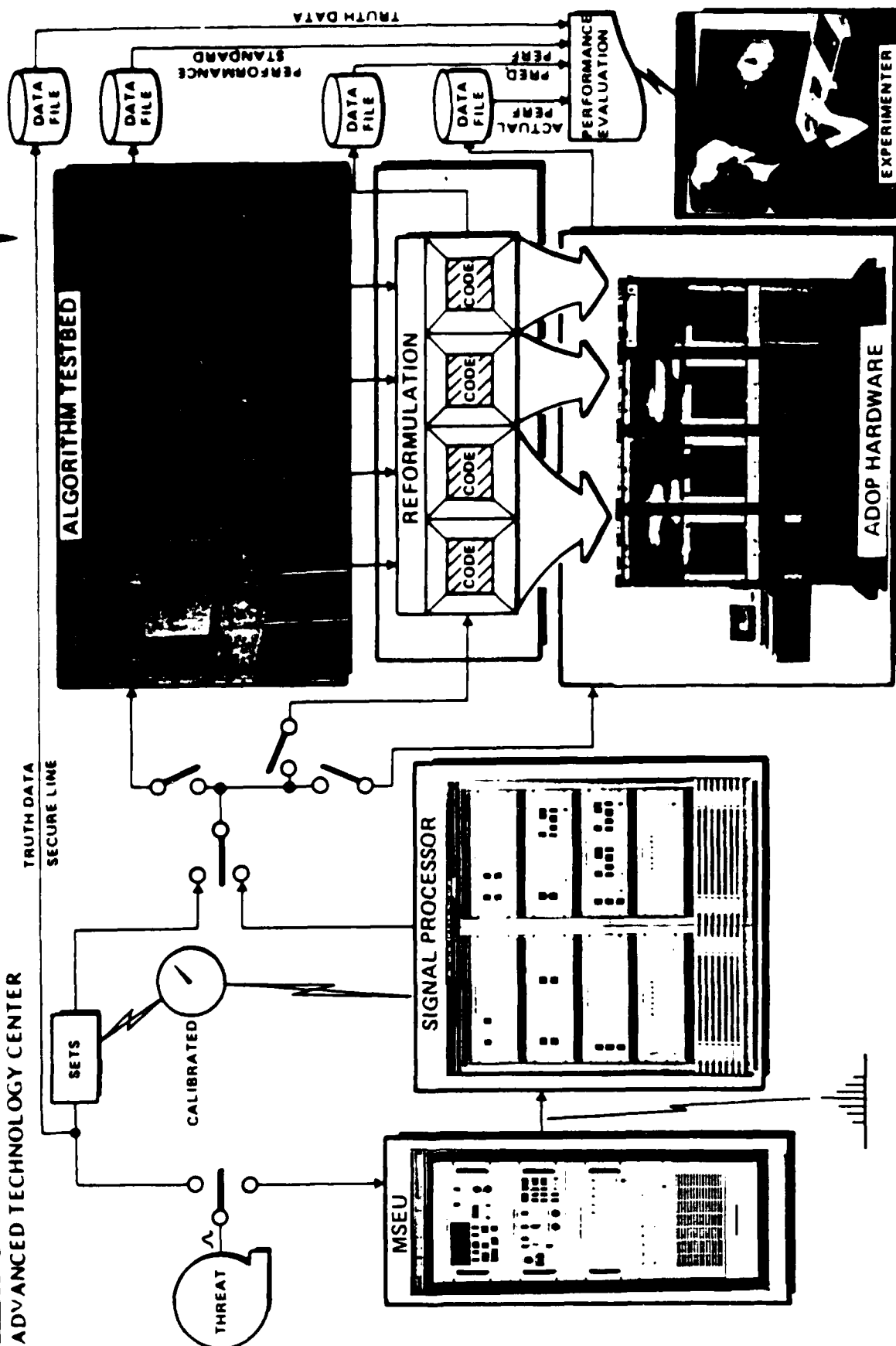


EXAMPLE CURRENT EVALUATION CAPABILITY

BMD

ADVANCED TECHNOLOGY CENTER

UNCLASSIFIED



On Parallelism in Software/Hardware Design Tools

P.A.Subrahmanyam
AT&T Bell Laboratories

Abstract. This paper explores opportunities for parallelism in some classes of computer aided design tools, and attempts to convey a few insights about how one might go about exploiting this parallelism. More specifically, perspectives that bear on the design of hardware accelerators for some important classes of design tools are provided; however, the actual details of such architectures are not specified.

Keywords: Parallel architectures, Design aids, integrated system design, reusable software.

1. Introduction

It is widely acknowledged that the design of software/hardware is complex, error prone, and therefore costly. Two broad classes of attempts to address this problem include various flavors of design methodologies and computer aided design tools. In this context, a "design environment" usually consists of some collection of tools supporting some combination of methodologies for the tasks concerned. While significant progress in design environments has clearly been made in the last few years, there is still an urgent need to make such design environments conceptually cleaner and faster. A naive way to improve existing execution speeds is to build special purpose hardware for each tool. This seems impractical; a major problem lies in fact that there are diverse concepts, terminology, and tools involved in a sophisticated design environment, and there is a tremendous cost overhead associated with designing special purpose hardware.

An alternative suggested here is that based on some notions of what is conceptually cleaner, and that are in consonance with emerging trends in formally based approaches to specification, programming languages and design paradigms, we can *partition* a design environment in a manner that potentially

- enables a better amortization of development costs for (a class of) specialized HW tools, and
- supports parallelism.

The rest of this paper elaborates mainly on these issues. In the next section, we summarize some of the major *activities* involved in the design process, suggest *paradigms* for carrying out these activities, and then focus on the *tools* needed to support these paradigms. The objective here is to highlight the fact a surprisingly large number of activities, although not all, can benefit from a common set of tools (provided that adequate attention is paid to the global architecture of such design aids). The importance of this observation is that it suggests a partitioning of the tools in a design environment that enables a much better amortization of hardware development costs for special purpose engines. The structure of the resulting design environment is then examined, with a view to exploring the potential for parallelism in its execution. We classify the various forms of parallelism possible, and indicate what flavors of hardware architectures may be suited to support the execution of advanced design tools of this nature.

A subliminal assumption made here is that formally based approaches to design - both methodologies and tools - have the advantage of enabling *provably correct* designs, and encourage the use of rigorous specifications at a high level of abstraction. In turn, this enables rapid prototyping of the specifications, and tends to reduce overall design and maintenance costs. Furthermore, the software tools used in such contexts are arguably conceptually clearer

and more robust: They are typically based on a standard infrastructure, i.e., one where the domain has been well studied (such as formal syntax and semantics of languages, algebra or logic); and tools (such as grammar-driven editors and symbol manipulation systems) have evolved that support typical operations in these domains.

The overall cost-effectiveness and robustness of design environments that employ such tools can be greatly improved if components of a design aid that are aimed at different activities can use a common pool of tools. Additionally, the potential advantages are further enhanced if there is some parallelism intrinsic to such systems that is amenable to intelligent exploitation, particularly via hardware accelerators. The main motivation for the research discussed in this paper is to explore design tools that possess such characteristics.

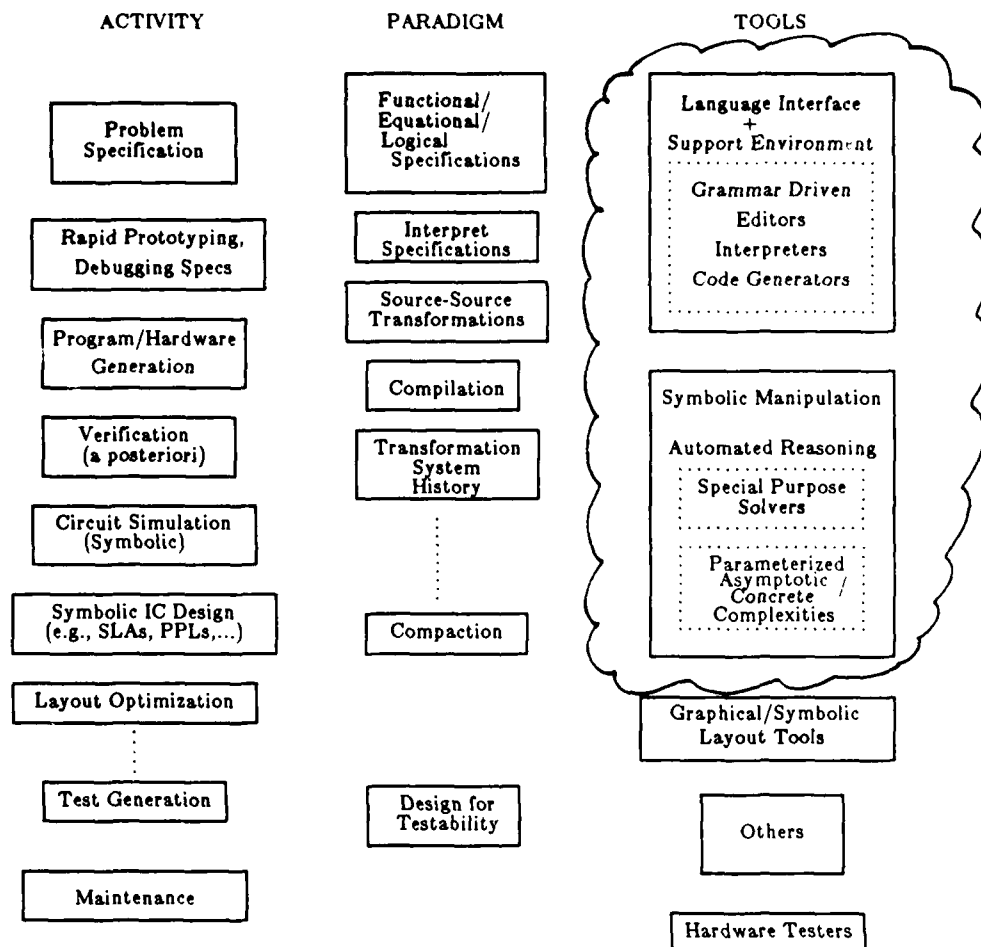


Figure 1. Design Activities, Paradigms, and Tools

2. Design: Activities, Paradigms, and Tools

Figure 1 indicates some of the major activities involved in the design process, suggest paradigms for carrying out these activities, and then focus on the tools needed to support these paradigms. This figure is designed not to be exhaustive, but merely to convey a flavor of some of the major activities involved in integrated system design. Our main intent here is to establish a context

for what follows, rather than to discuss involved in great depth.

The activities indicated in Figure 1 include problem specification, rapid prototyping of specifications, program or hardware generation, and maintenance. Problem specification involves an explicit statement of the problem in a programming language. While this can in principle be any programming language, there are cogent arguments as to why this should be a high level language with a declarative flavor, and more specifically one that minimizes the amount of representation dependent information that must be specified.[3] In view of this, the mode of specification depicted suggests the use of functional, equational or logic based techniques.[4]. Good data structuring and knowledge representation capabilities are also needed. These are being supported in the context of evolving systems in the context of logic and relational systems[2].

To help "debug" the specifications, facilities for rapid prototyping are needed. This implies the need for support tools like syntax and semantics based editors, interpreters and symbolic execution facilities. Such tools are increasingly driven by the syntax and semantics of the language grammar as opposed to being custom coded.

The next phase usually involves the generation of either software or hardware that consistently implements these specifications. This may be done using traditional compiler techniques, or a somewhat generalized form of compilation that may be conceptualized as transformation based techniques. The actual generation of hardware designs involves other lower level activities such as symbolic IC design, cell layout, simulation at various levels, a posteriori verification, testing etc. In both cases, the tools needed include term (or tree) manipulation systems, semantics driven tools. Source-to-source transformations additionally need various forms of formal manipulation or term rewriting systems, along with limited kinds of theorem proving abilities. Strategy guidance aspects of such systems need the ability to deal with performance or complexity measures, and this in turn relies on symbol manipulation tools like those found in Macsyma, for instance. Other activities such as verification, simulation, layout optimization, and maintenance, also benefit from tools of this flavor.

Our main intent here is to emphasize the somewhat central role of this class of tools, should they exist, rather than to suggest an exclusive set of tools and paradigms. A consequence of this observation is that while there are diverse tasks involved in the overall system design process, there seems to be a set of tools that is frequently used by a fairly broad spectrum of these activities.¹ Of course, this observation is not entirely new, but given the emerging programming paradigms that merge object based and logic programming,[2] as well as the interest in using similar techniques in the hardware domain,[7] we believe that the existing commonalities are much broader than was perhaps originally recognized.

In view of this observation, we will next comment on the potential for parallelism in such a context, and attempt to provide a basis for discussing the design of hardware accelerators for such tools.

3. Parallelism in the Implementation

The overall parallelism in a design environment can stem from essentially three main sources, since we can consider

1. The specific tools in this kernel is expandable, and will depend upon the paradigm adopted for addressing each of these activities.

- Parallelism in the system (design environment) structure
- Parallelism in the underlying tool structure
- Parallelism in the language support structure, i.e. the implementation of the language the tools are written in

In discussing hardware support for such parallelism, we will, for simplicity, consider two broad classes of such parallelism that are achievable via

- tightly coupled processes (shared memory architectures) and via
- loosely coupled processes (message-passing-based architectures)

We first comment on the potential parallelism at the level of abstract processes that arise out of the problem structure. We then look at the attributes of this abstract partitioning in terms of possible implementations using the two dominant paradigms for building hardware, namely shared memory and message-based architectures

3.1 Parallelism in the System Structure

We first consider the parallelism in the structure of a design environment. Each of the activities in the "activity" diagram may be executed "concurrently" on a loosely coupled system. In fact, this can be done in several cases in a pipelined or "systolic" sense, because of the input/output dependencies involved. Some of the activities themselves may be parallelized e.g. simulation of large networks. Since a large number of activities share a common set of tools, additional processing power is exploitable via duplication of tools. The basic parallelism available at this level is that exploitable by multiple users timesharing hardware accelerators that support a common kernel of tools

3.2 Parallelism in the underlying tool structure

To investigate parallelism in the underlying tool structure, it is obviously necessary to consider each tool individually. As an example, we consider here an important but somewhat non-standard tool: clause based automated reasoning systems, and languages based on logic, functions, and equations.

The rationale for this choice is the following:

- "Rule based-Expert Systems" are becoming a popular programming paradigm; logic programming provides a cohesive formal basis for such systems
- A number of research efforts are attempting to enrich the logic programming framework with the abilities to do knowledge based reasoning. Except for theological differences, such systems show signs of soon becoming competitive with Lisp-based environments.
- An orthogonal set of research efforts is aimed at integrating the notions of logic, functional, equational, and object-based programming.[2, 4]
- "Common sense reasoning" techniques need to be supported by such tools.
- Clause based systems have been receiving special attention because of the success of Prolog.
- A number of the tools e.g. rewrite systems, simplifiers, complexity calculations, etc. can potentially be expressed relatively easily using such systems.

Of course, there are a number of technical issues to be resolved in many of these areas. The intent here is not to suggest that everything is either simple or has been solved, but merely to establish a conceptual framework for discussing exploitable parallelism in such systems.

3.2.1 Clause-based systems. Most clause based systems incorporate a central component similar to that shown in Figure 2. The basic function of each of the components in the diagram is as follows:

1. *Choose a single clause* from the knowledge database. This clause is used, along with other clauses, to infer new clauses. The choice of this clause is usually governed by some criterion that determines how relevant or "promising" this piece of information is.
2. *Generate new clauses* using the given clause, a variety of inference rules, and (optionally) other clauses in the knowledge base.
3. *Simplify Inferred Clauses* involves the reduction of the inferred clauses to a normal form. This process often requires a fair amount of computation, with a minimal amount of communication involved.
4. A *Generality Check* is made to check whether the newly inferred clauses already exist in the knowledge base.
5. One the new clause is deemed useful, it needs to be *integrated into the knowledge base data structures*.

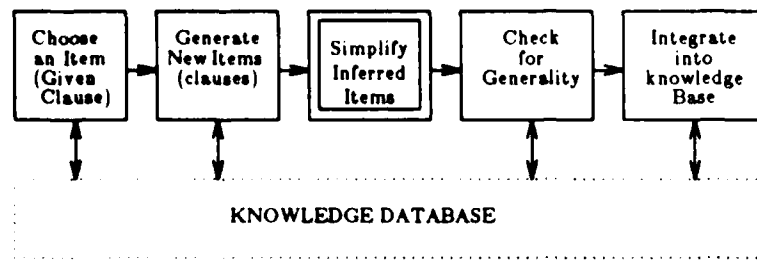


Figure 2. Subprocesses in a clause-based theorem proving system

In such a system, the simplification process can usefully be run on a separate processor (it is compute bound, and requires a low bandwidth of communication with the knowledge base). All of the other processes typically interact heavily with the knowledge base. The basic requests that arise are of the form that require the set of formulas that satisfy a boolean conjunct, and that can be unified with a specified formula. The degree of useful parallelism (loosely coupled) is determined by how expeditiously the data base accesses can be resolved.

An efficient access technique, indicated in Figure 3, is to partition the clausal knowledge base into roughly two parts:

1. The raw data is represented as a very compact list structure, in which each formula occurs just once, with "pointers" relating it to each occurrence in some superstructure.
2. A second structure is constructed to offer rapid access to raw data. It is a set of inversions based on distinct properties of the formulas, allowing rapid isolation of the desired formulas.

An important characteristic of this representation is that the access structures can be easily partitioned into an arbitrarily large set of substructures. The set of formulas can be partitioned into an arbitrarily large number of subgroups, with each group accessed through a different access structure. Unfortunately, the core data does not seem amenable to partitioning as easily.

The clause-based component of a design environment may therefore be viewed as a number of such pipes running in parallel, all interacting with the shared knowledge base and coordinated by a single resource manager. The possible response time is then restricted by the response time of the knowledge base. The hardware support appropriate for the process level partitioning described above would consist of a loosely coupled hardware engine for performing the simplifications, together with a tightly coupled shared-memory system that supports the execution of the remaining processes.

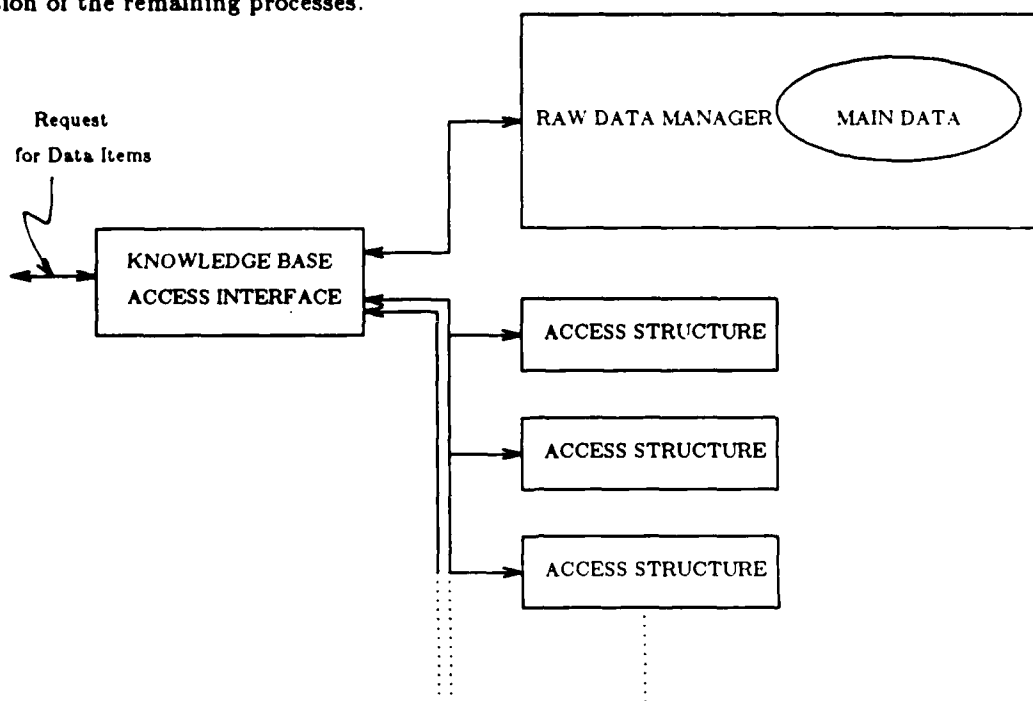


Figure: Partitioning access structures to the knowledge base.

3.3 Other sources of Parallelism

As indicated above, many of the tools in the design environment may be amenable to parallelization; however, each must be examined on an individual basis. Currently, research along such lines is in its very early stages, with the greatest emphasis being placed on hardware accelerators for circuit simulation[6]. There are several ongoing research efforts that attempt to exploit parallelism in circuit simulation at various levels of detail, for example, by partitioning circuits across multiprocessor clusters. In addition, there are an increasing number of commercial products that are now available that address the simulation task. Additionally, hardware accelerators for routing are also being studied as a means of improving the performance of VLSI design tools, e.g., [5]

3.4 Parallelism in the Underlying Language Implementation.

A further source of parallelism in the design environment is parallel support for the languages that are used to implement the tools. This particular task clearly has a very broader appeal, and consequently there are a number of efforts aimed at providing multiprocessor implementations for functional and logic languages, as well as for Algol- and Lisp-like languages. In this context, functional languages, in expressing computations as the evaluation of

expressions such as $f(t_1, \dots, t_n)$ allow for a very natural expression of the concurrency that is possible in an execution: t_1, \dots, t_n can all be executed in parallel. This observation extends to many of the symbolic manipulation operations involved in logic and algebra based systems. Consequently, this is an area where any progress will have significant impact.

4. Summary

This paper has highlighted the fact that a surprisingly large number of software and hardware design activities can, given appropriate paradigms, benefit from a common kernel of tools. The nature of these tools was identified, and different sources of parallelism in the implementation of design environments for software/hardware systems were examined. We envision that future generation design environments will have increasing degrees of hardware support for such a kernel of design tools, as well as the underlying implementation languages. The identification of the components in such commonly used tool sets will help better amortize the development costs involved in building special purpose hardware accelerators for such tools. The existence of such advanced design environments will, in turn, serve to improve the cost-effectiveness and robustness of systems designed using these tools.

References

- [1]. Lusk, E.L. and Ross A. Overbeek, *The Automated Reasoning System ITP*, Argonne National Laboratory, Argonne, IL (1984).
- [2]. F. Mizoguchi, K. Furukawa (Guest Editors), "Special Issue on Knowledge Representation," *New Generation Computing* 3(4)(1985).
- [3]. J. A. Goguen, "Parameterized Programming," *IEEE Trans. on Software Engg.* SE-10 pp. 528-552 (September 1984).
- [4]. D. DeGroot, G. Lindstrom (Eds.), *Logic Programming: Relations, Functions, and Equations*, Prentice-Hall (1986).
- [5]. S.J.Hong and R.Nair, "Wire Routing Machines - New Tools for VLSI Design," *Proc. IEEE* 71(1) pp. 57-65 (Jan 1983).
- [6]. R. Smith, "Fundamentals of Parallel Logic Simulation," *Proc. 29rd Design Automation Conference*, pp. 2-12 (June 1986).
- [7]. G. Milne and P. A. Subrahmanyam (Eds.), *Formal Aspects of VLSI Design*, North Holland, Amsterdam (1986).

Session 5: Interconnection Strategies

Chairperson: J. Richard Burke
Research Triangle Park

Role of Broadcasting in Multiprocessor Systems

Binay Sugla
Sudhir Ahuja
AT&T Bell Laboratories
Computer & Robotics Research Laboratory
Holmdel, New Jersey 07733

I. Introduction

Broadcasting presents a powerful technique for parallel/distributed computing. As of now it remains a largely unexplored territory especially in terms of the impact it may have in all areas of parallel processing. In this context we summarise our experiences with broadcasting in the several multiprocessors built over the last few years at our research laboratory. It has been found that broadcasting can play an important role not only in terms of improving algorithmic efficiency but also with respect to making the parallel programming environment more powerful. In order to substantiate this conclusion we present the role of broadcasting in the design and implementation of parallel algorithms (e.g. prefix computation) and in the implementation of the concurrent programming language Linda.

II. The Model of Broadcasting & Previous Research

In the weakest form of broadcasting one processor transmits one message on the bus per unit time. During this time all the other processors listen to the broadcast. Translating this concept into that of a shared memory model this implies a capability on the part of processors to read any one memory cell simultaneously.^[1] In the parallel circuit model this would permit processing nodes to have unbounded fan-out - a capability which results in reduced complexity of parallel computation for some problems like prefix computation.^[2] Even though capabilities of broadcasting have relatively been ignored in literature some efficient algorithms (e.g. finding an extremum, sorting, selection, merging) based on the mechanism of broadcasting have been proposed.^{[1][3]} Here we present our experience with broadcasting as a useful medium for implementing a wider class of algorithms (e.g. prefix computation) and the concurrent programming language Linda. The multiprocessor considered here is the S/NET.^[4]

The discussion will also attempt to bring out the different variations of broadcasting that are possible. For our purposes let the channel be slotted into slots of time T_B . Each of these slots any one of these processors may execute a broadcast. Thus if the bus connecting N processors is N times as fast as the processor cycle time T_P it has the potentiality of letting each processor broadcast one message during one processor cycle time $T_P = N * T_B$. This infact describes the S/NET bus. In the ensuing discussion the questions of synchronization are dealt with at the algorithmic level - that is, the algorithms considered are synchronous in nature. The issue of bus contention is presumed to be decided at the hardware level as in S/NET.

III. Applications

As mentioned before the impact of the broadcasting mechanism are fairly wide spread - from specific computational problems to implementation of parallel programming languages like Linda. In this section some of these applications are summarized while the others are mentioned.

A. Prefix Computation

Given N inputs $x_1, x_2, \dots, x_{N-1}, x_N$ and an arbitrary associative operation \circ the problem of prefix computation is to produce the N outputs $x_1 \circ \dots \circ x_i$, where $1 \leq i \leq N$. This problem appears in hidden forms in many commonly occuring computations, for example, evaluation of

linear recurrences, carry computation in binary adder etc.² If we have N processors with one input in each processor it may be deduced that time of computation is $(T_p + T_b) \cdot N$ which is of the same order of complexity as that for a sequential processor. If however we have only K processors the time taken can be reduced to $T_p \cdot (N/K + K) + T_b \cdot K$. The algorithm is as follows. Each processor computes the the prefix problem on its N/K inputs. The last prefix from processor i is broadcast which is used by processor $i+1$ to compute immediately its last prefix which is then broadcast. While the values of the last prefix are being communicated between processors i and $i+1$ the processors $1 \dots i-1$ are busy calculating the lower order prefixes. The algorithm assumes that the inputs are distributed in a prescribed manner - processor i contains the inputs $x_{N(i-1)/K + 1} \dots x_{N/K}$. Thus the best time for the case $T_b \leq T_p$ is achieved when $K = \sqrt{N}$.

Interestingly, however, a simple variant of this algorithm works on *any* distribution of data. To see how this algorithm works assume any data distribution of x_1, \dots, x_N among the K processors available. Also for convenience describe x_1, \dots, x_N as $[i, j]$. In this notation then, given $[i, i]$, $1 \leq i \leq N$ we are asked to compute all $[1, i]$. By $\max [i, j]$ we mean x_1, \dots, x_k , such that $k = \text{maximum } j \text{ for which } x_1, \dots, x_k \text{ is available or can be formed from the locally available values of data.}$ The sketch of the algorithm which runs on each processor is as follows.

((If processor i contains $[i, i], [j, j], [k, k]$.. then
it is also responsible for calculating $[1, i], [1, j], [1, k]$. All
processors execute the same algorithm).

while there exists a unused data value do
 pick an unused $[i, i]$ with least index i
 compute $\max [i, j]$
end of while

Broadcast max [1, i]

while an [1, i] remains to be computed do
 receive $[1, j]$
 compute $\max [1, k], k > j$
 broadcast $\max [1, k]$
 compute all computable $[1, m], j < m < k$
end while }

In fact stronger properties about the performance of the algorithm can be shown. If we define the parallel time of computation such that it includes the time taken to redistribute the data as well as the actual computation time then it may be proved that the algorithm achieves optimal speedup for a large class of data distribution. Thus the gains from having a desirable data distribution may be offset by the time taken to achieve that distribution. It may also be noted that feature of broadcasting is instrumental in making this algorithm work for all data distribution.

B. Linda Implementation

The concurrent programming language Linda was designed at Yale University and implemented on the S/NET by Carriero et. al.^[6] Linda differs from other parallel languages by adopting the generative communication model. A shared tuple space (TS) is accessible to all the processes in the distributed program and two processes may communicate only by reading from or writing into this shared tuple space (Figure 1). The broadcast capability provides a convenient underlying communication mechanism to implement the language. There are four operations defined over TS: *out()*, *in()*, *read()* and *eval()*. *out(s)* adds a tuple t to the TS, *in(s)* withdraws a matching tuple t from TS, *read(s)* assigns actuals to the formals as in *in(s)* except

that the tuple is not removed from TS. $eval(s)$ adds an unevaluated tuple to TS. This design feature of the language offers enhanced efficiency in a parallel programming environment where data objects may have to be shared. Using the facility of broadcasting this TS is implemented in a simple manner as follows. The tuple space is duplicated on all processors. Whenever an $in()$, $out()$ etc. is executed the corresponding template is broadcast and a similar action is taken by all the nodes. In this way the synchronization (and thus the consistency of TS in all processors) is effectively realized. A second implementation in which the tuples remain on the originating nodes until they are explicitly asked for is currently being implemented^[6]. The functions $in()$, $out()$ etc. are then realized by broadcasting the templates - the corresponding action of insertion or deletion from the TS is taken by the processor which contains the matching tuple.

IV. Conclusion & Research Issues

It should be clear that broadcasting has many innovative uses which should be further explored. These uses can be classified into fundamental (that is, algorithmic) and implementational. For example, the use of broadcasting in implementation of multiprocessor operating systems offers some advantages.

Another issue of significant interest is the relative advantages and disadvantages which multiprocessors with different T_B & T_P have to offer. For example, broadcasting on a hypercube^[6] of N nodes takes time $\log N$. This may be contrasted with the fast speed of the S/NET bus in which the time taken per broadcast is $1/N$ times the processor speed. But again if we conceive of multiple broadcasts taking place simultaneously - as in the hypercube, interesting applications arise. Another related issue is the use of broadcasting in absence of synchronization among the messages belonging to a single broadcast.

In summary, broadcasting offers interesting possibilities not only in the execution of specific algorithms but also in the implementation of general parallel programming software and as such will play an important role in future multiprocessing systems.

REFERENCES

1. Dechter, Rina and Kleinrock, Leonard, "Broadcast Communications and Distributed Algorithms," *IEEE Transactions on Computers*, Vol. C-35, No 3, March 1986, pp 210-219
2. Sugla, Binay, "Parallel Computation Using Limited Resources," Ph.D. Dissertation, Department of Electrical and Computer Engineering, University of Massachusetts at Amherst, July 1985.
3. Levitan, S., "Algorithms for broadcast protocol multiprocessor," *Proc. 3rd Int. Conf. on Distributed Comput. Syst.*, 1982, pp. 666-671.
4. Ahuja, S. R., "S/NET: A High Speed Interconnect for Multiple Computers," *IEEE Transactions on Selected Areas in Communications*, SAC-1, 5, November 1983
5. Carriero, N. and Gelernter, D., "The S Net's Linda kernel," *in Proc. of the Conference on Principles*, December 1985
6. DeBenedictis, E., "Multiprocessor Programming with Distributed Variables," *in Proc. of the Conference on Hypercube Multiprocessors*, August 1985

AD-A184 949

PROCEEDINGS OF THE WORKSHOP ON FUTURE DIRECTIONS IN
COMPUTER ARCHITECTURE. (U) BATTELLE COLUMBUS LABS
RESEARCH TRIANGLE PARK NC D P AGRAWAL ET AL 30 AUG 86

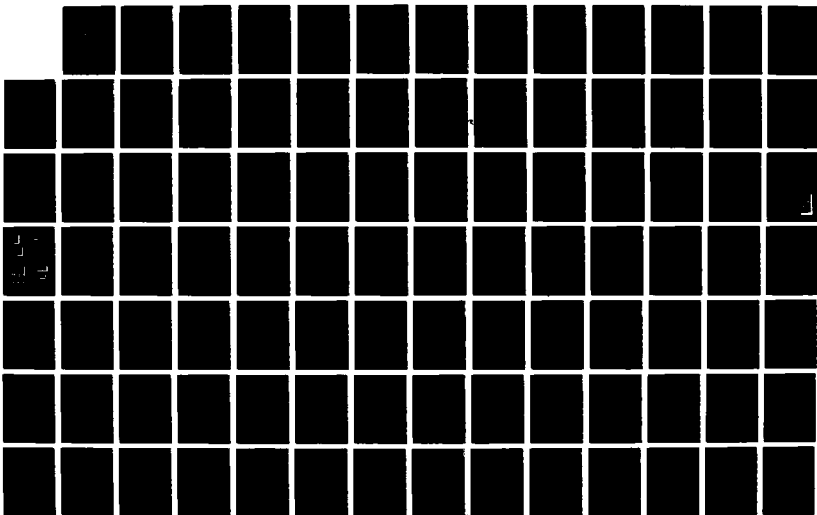
2/5

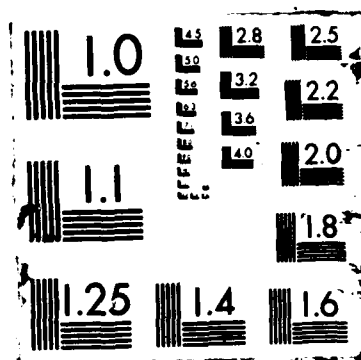
UNCLASSIFIED

ARO-86384-EL DARG29-81-D-8100

F/G 12/5

NL





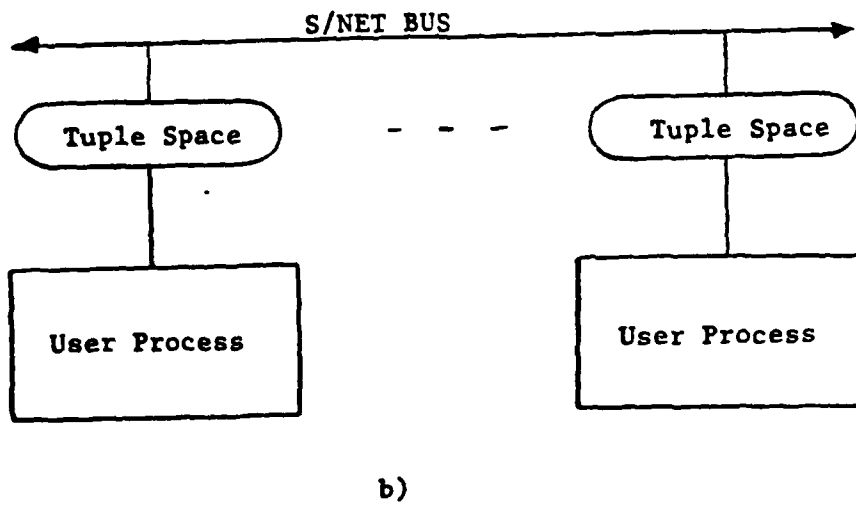
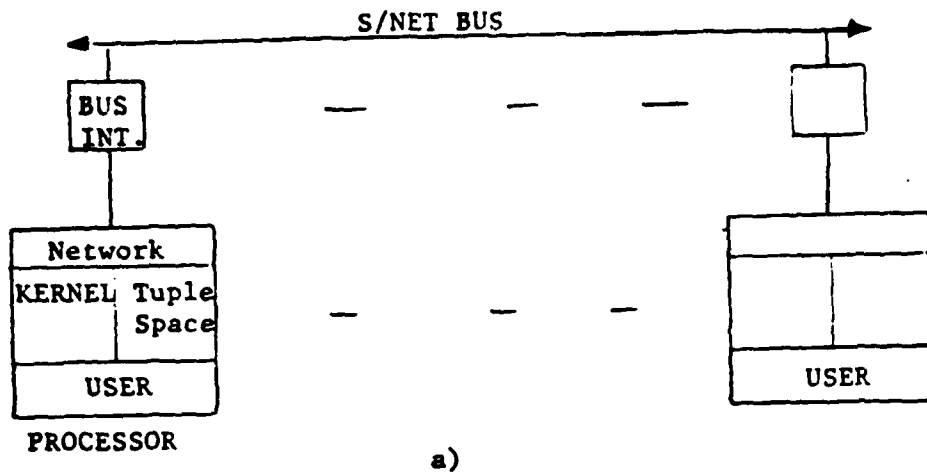


Figure 1: LINDA on S/NET

Image texture classification with an optical crossbar interconnected processor.

Alastair D. McAulay

Texas Instruments, Central Research Laboratories, P.O. Box 226015, MS 238.
Dallas, TX 75266.

INTRODUCTION

Efficient use of extensive parallelism with a wide range of algorithms is required to meet future computation demands. The problems are discussed of providing: high performance, flexibility, extendability, reliability and ease of programming. An optical crossbar interconnected processor is described, including the associated optics, and programming methodology. The Levinson-Durbin algorithm for solving Toeplitz matrix equations is considered as an illustration of how 2-D autoregressive models could be computed for image texture classification. An executable flow graph is generated for the Levinson-Durbin algorithm with unrolled iterations. The flow graph is implemented on a simulator and the resulting activity chart shows a 100% efficiency once the pipeline is filled. The advantages and limitations of the proposed architecture are summarized in the conclusions.

DIFFICULTY OF ACHIEVING DESIRED FEATURES.

Extendability implies that more processors may be added to the multiprocessor together with corresponding interconnections without requiring new software and with performance approaching linear improvements with number of processors. The number of parallel processors that may be used efficiently is limited in today's prototype and proposed systems by the communication delay and interconnection complexity. These systems are not generally very extendable.

Flexibility to efficiently run a wide range of algorithms may be achieved by reconfigurability. Reconfigurability also enables software to readily adjust to an extended system. However, reconfigurability introduces large delays and high control overhead in most proposed systems. This severely restricts the number of processors that may operate efficiently.

Nearest neighbor interconnected configurations are easy to construct and provide high speed communications. The number of elements in an array may be increased without altering the bandwidth at an input or output connection. This permits large numbers of processors to be efficiently used in parallel for suitable algorithms. However, the nearest neighbor connections limit the flexibility or range of algorithms that may be implemented efficiently. For example, many fast algorithms use

recursive doubling which results in complex non nearest neighbor communications, e.g. FFT. Systolic systems prearrange dataflow so that input and output occurs at the edges of the processor array at each cycle ². Latency is increased relative to serial machines and this causes difficulties for general purpose or adaptive computing.

Reliability is often accomplished by means of redundancy in software, hardware, time and/or space. This is detrimental to satisfying performance for a given cost. Extended systems are likely to be less reliable because of the greater possibility of interference.

Ease of programming is important. Mapping a complicated algorithm flow graph to a complicated machine flow graph is difficult. The ability to perform the mapping automatically is enhanced if the machine flow graph is less constrained, as for a crossbar interconnected system.

PROPOSED OPTICAL CROSSBAR INTERCONNECTED PROCESSOR

System

Figure 1 shows a preliminary organizational structure for an optical crossbar signal processor ^{4,3}. The inputs and outputs of hundreds of processing elements are connected to an optical switch by means of commercially available fiber optic links of bandwidth 160 MHz or more. The processors perform elementary operations such as multiply or add and therefore have two input connections for the two operands. This fine granularity permits the maximum amount of parallelism to be extracted from algorithms. The processing element output is converted from parallel to serial in a shift register for driving the fiber optic link. A second fiber optic loop between processors and main memory banks provides input/output.

Optical switch and interconnections

Optics is used for interconnections because, relative to electronics, it provides high levels of parallelism, high bandwidth, large fan-in and fan-out, and high immunity to interference. These features enable the use of fine granularity and reconfigurability. Each intersection in a crossbar switch, Fig. 2a, has a switch permitting a horizontal input line to be coupled with a vertical output one. Figure 2b shows a diagrammatic crossbar switch implemented with a spatial light modulator (SLM) and dots indicate transparent regions consistent with the closed switch settings marked by dots in Figure 2a. An optical lens system is used to spread the light from the input sources horizontally without spreading the light vertically. Light passing through the spatial light modulator is collapsed on to receiving diodes by means of a lens system which focusses vertically without spreading horizontally.

A reflecting membrane covers the surface of an array of transistors on a silicon chip, Fig. 3, to form a deformable mirror device (DMD). The right side of the modulator is folded back and a beam splitter used because DMD's are reflective SLM's. Activation of a transistor causes the membrane to dip above the transistor. A Schlieren system accounts for the reflections from the regions between mirror deflection pixels. Imaging and performing spectral analysis with a Texas Instrument's DMD of size 128 by 128 has been published [5]. It takes one microsecond per row to set the DMD up and a few microseconds to switch to the new settings.

Software and programming approach

The flow of data is prearranged so as to minimize run time overhead. A compiler maps an algorithm graph into the processor: assigning the nodes in the graph to processing elements and the edges or links to crossbar settings. Data flowing into the switch is routed to the appropriate processor. A processor will perform the operation for which it is programmed on the next clock cycle after receiving its operands. The output is routed via the switch to the next processor.

EXAMPLE OF PROGRAMMING AND PERFORMANCE EVALUATION

Image texture classification example

The idea is to compute, for each segment of an image, a 2-D AR model or filter that when applied to the 2-D image segment will remove all the information leaving only white noise. The 2-D filter may then be considered an estimator of texture. It may be used to correlate against templates for the problem domain, e.g. trees and roads. 2-D dynamic programming may be appropriate to allow for stretch or compression. Wiener or least square filter theory [1] may be used to derive a set of block Toeplitz matrix equations whose solution are the required filter.

Yule-Walker equations and Levinson-Durbin algorithm

In order to simplify the discussion the 1-D case is considered. The implementation of the 2-D case on the proposed processor is thought to be a straight forward extension. The Yule-Walker equations may be solved for 1-D AR parameters a from

$$\begin{bmatrix} r_0 & r_1 & \cdot & \cdot & r_{m-1} & r_m \\ r_1 & r_0 & \cdot & \cdot & r_{m-2} & r_{m-1} \\ \cdot & & & & & \cdot \\ \cdot & & & & & \cdot \\ r_{m-1} & r_{m-2} & \cdot & \cdot & r_2 & r_1 \\ r_m & r_{m-1} & \cdot & \cdot & r_1 & r_0 \end{bmatrix} \begin{bmatrix} 1 \\ a_1 \\ \cdot \\ \cdot \\ a_{m-1} \\ a_m \end{bmatrix} = \begin{bmatrix} r_m \\ 0 \\ \cdot \\ \cdot \\ 0 \\ 0 \end{bmatrix} \quad (1)$$

r_τ is an estimate of the autocorrelation function at lag τ and the mean is μ .

$$r_\tau = \frac{1}{N} \sum_{j=1}^{N-\tau} (x_j - \mu)(x_{j+\tau} - \mu) \quad \tau = 0 \text{ to } m. \quad (2)$$

The Levinson-Durbin algorithm follows. At the n th iteration a reflection coefficient is computed as the inner product

$$c(n) = \sum_{i=0}^{n-1} \frac{a_i r_{n-i}}{v(n-1)}. \quad (3)$$

The power of the white noise associated with the AR process is computed from

$$v(n) = v(n-1)(1 - |c(n)|^2). \quad (4)$$

Minimum delay is maintained by updating the AR parameters from

$$a_k(n) = a_k(n-1) - c(n)a_{n-k}(n-1) \quad k = 0 \text{ to } n. \quad (5)$$

The autocorrelation function may be computed by reconfiguring the crossbar and processors to represent a tree 4.

Flow graph for unrolled Levinson-Durbin algorithm

The processing of many segments of an image provides continuous sets of data for which the algorithms must be performed, therefore pipelining is advantageous. The Levinson-Durbin iterations are unrolled into a long section of code. Only four loops are unrolled to simplify the explanation.

In the flow graph, Figure 4, each operation is assigned to a node of the graph and the flow of data between operations is identified by links between nodes. Nodes marked with subtraction imply the subtraction of the right hand input from the left hand input. The triangular arrows indicate negation or unary minus which may be accomplished at the input to the appropriate node rather than with an extra node. Identity instructions marked "Ident" support the fanout of operands so that an input connects to a processing element before going through the crossbar. Delays are inserted as numbers on the graph edges.

Performance of Levinson-Durbin's algorithm

Figure 5 shows an activity chart resulting from running a dataflow simulator for the flow graph of figure 4. Processors 1 and 2 operate in parallel in the first Levinson-Durbin computation. In clock 2 the processor node 3 now operates on this computation. Meanwhile processors 1 and 2 are free to start on the next Levinson-Durbin computation. After 11 clock cycles the pipeline is full, and the

result for the first Levinson-Durbin computation is obtained. Once the pipeline is filled, the efficiency of computation is 100%. It is reasonable to unroll up to 10 iterations of Levinson-Durbin for operation with the 500 processor system and still have sufficient processors for the other computations necessary.

Conclusion

Advantages of proposed processor. Fine granularity processing elements permit the efficient use of high levels of parallelism. Flexibility to run a wide range of algorithms with ease is achieved by reconfigurability. The number of wires connecting to an electronic crossbar switch of equal throughput to the proposed optical crossbar would be tremendous because parallel connections would be required. Optics has the bandwidth to permit serial connections. The processor may be doubled in size by adjoining a similar system and adding exchange switches. Programming may be accomplished simply by writing flow graphs and using a compiler to map the flow graphs to the machine. An image texture classification example was discussed to illustrate the programming procedure. Reconfigurability and optical interconnections improve reliability.

Limitations of proposed processor. The desirable features listed above are achieved by having a more costly interconnection system than, for example, a nearest neighbor scheme. Consequently, the proposed system is limited to high performance systems where flexibility, extendability and reliability are required. The large fan-out possible with optics by means of lenses is accompanied by an energy loss which increases the cost of the amplifiers required. The times to load and switch the crossbar switch constrain the rate at which algorithms may be changed in a dataflow mode.

Acknowledgments

Office of Naval Research and DARPA support under contract N00014-85-C-0755 is gratefully acknowledged. I wish to thank Jeff Sainpsell and Don Oxley of Texas Instruments for valuable discussions.

References

1. Justice J.H., "A Levinson-Type algorithm for two dimensional Wiener filtering using bivariate Szego polynomials," Proc. IEEE, Vol. 65, (1977.)
2. Kung H.T., "Why systolic architectures?," Computer, Vol. 15, pp. 37-46, 1982.

3. McAulay A.D., "Optical Interconnections for Real Time Symbolic and Numeric Processing." Chapter in Book "Optical Computing: Digital and Symbolic". Marcel Dekker. 1987.

4. - "Optical Crossbar Interconnected Signal Processor with Basic Algorithms." Opt. Eng., Vol. 25, pp. 82-90. 1986.

5. Pape D.R., and Hornbeck L.J., "Characteristics of the deformable mirror device for optical information processing." Opt. Eng., Vol. 22, pp. 675-681. 1983.

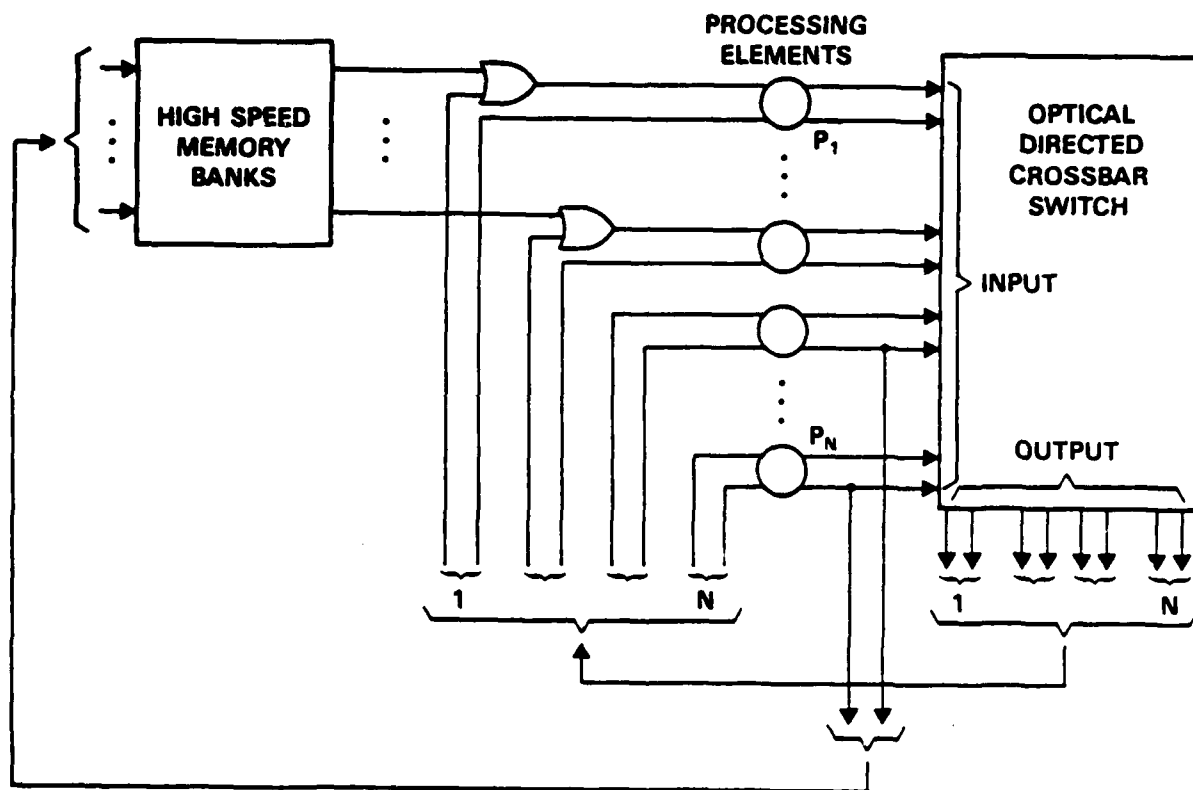
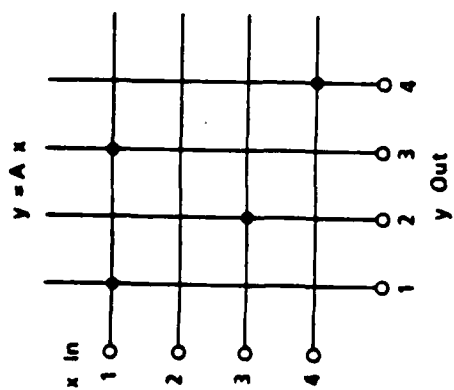
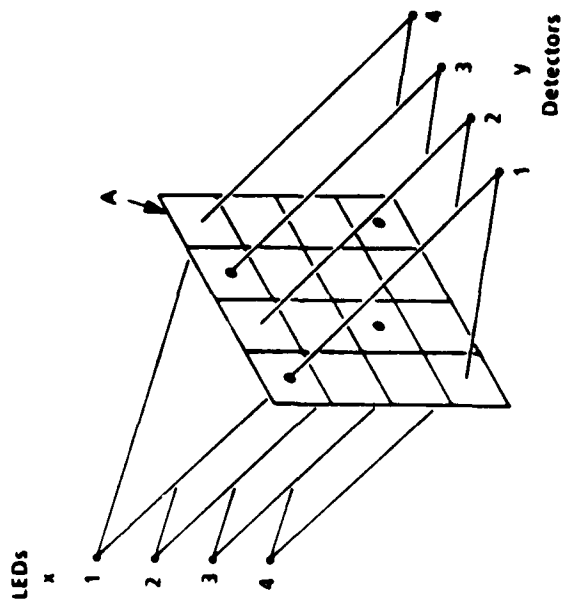


Figure 1: Organizational structure for optical crossbar signal processor



(a) Switch Settings



(b) Spatial Light Modulator Crossbar Switch

Figure 2: Diagrammatic optical crossbar switch

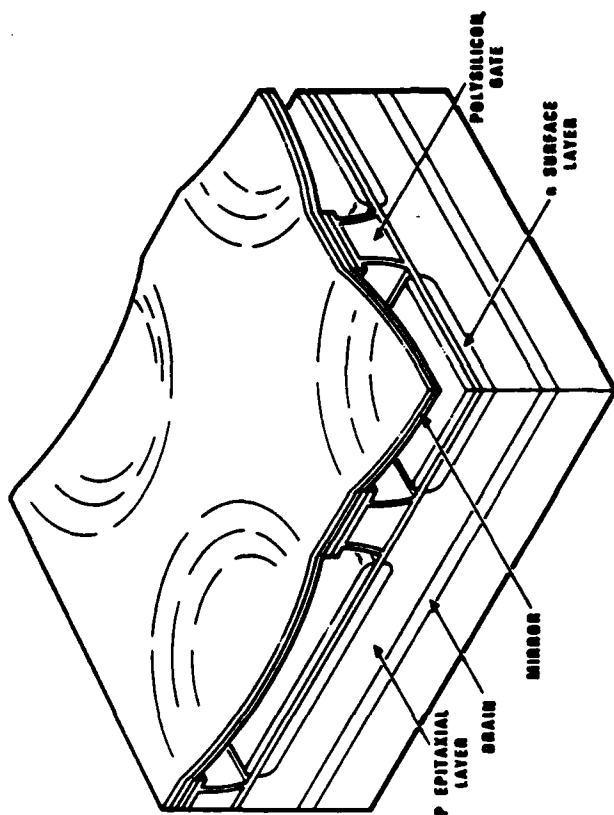


Figure 3: Deformable mirror device

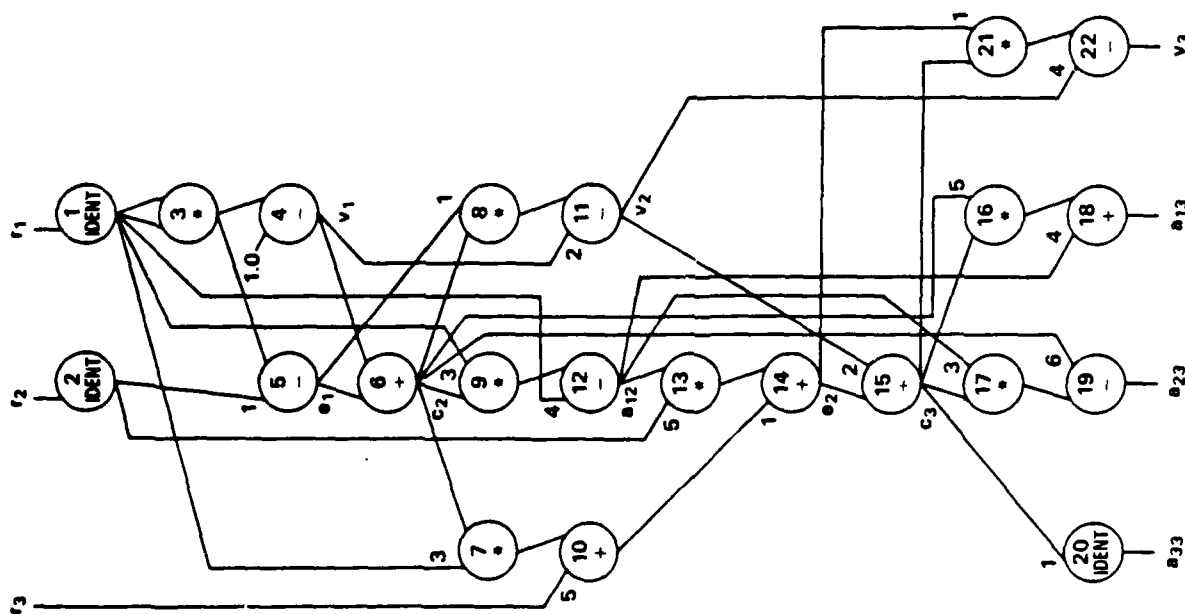


Fig. 4: Flow graph for unrolled Levinson algorithm

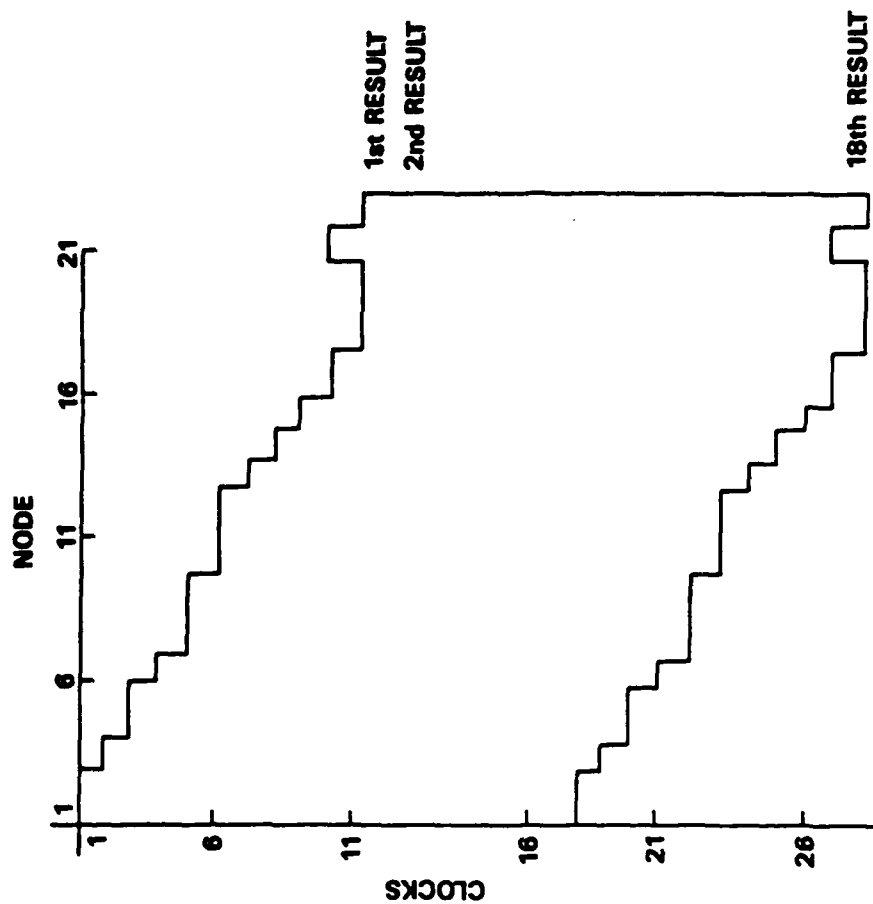


Figure 5: Activity chart for unrolled Levinson algorithm

MULTIPLE-BUS INTERCONNECTION FOR FUTURE MULTIPROCESSOR SYSTEMS

**L. N. Bhuyan
The Center for Advanced Computer Studies
University of Southwestern Louisiana
P. O. Box 44330
Lafayette, LA 70504**

SUMMARY

The performance of multiprocessor systems depends largely on the performance of the interconnection network (IN) that connects processors to memories or processors to processors. In addition to providing full connectivity at low cost, the IN must be fault tolerant and be suitable for a varied class of algorithms that the system may execute. Although a lot of emphasis has been placed lately on a class of networks called multistage interconnection networks (MIN's) [1], they do not satisfy the later conditions. Recently there has been an upsurge of interests in multiple-bus systems. This system provides inherent simplicity of the single-bus architecture while providing a large bandwidth (BW) for data transfer. Also, it is suitable for any kind of algorithms and has an added advantage of fault-tolerance. This paper will present the performance of both centralized and decentralized multiple-bus interconnections.

A multiprocessor organization can be broadly divided into two categories, namely : tightly coupled and loosely coupled. The tightly coupled multiprocessors are characterized by a close interaction between the processors that is effected through a bunch of global memories. In loosely coupled systems, the interactions are less frequent and there may not be any global memories for direct access at all. In a multiple-bus multiprocessor, there are B buses that connect N processors to N memories in a tightly coupled system or connect N processors in a loosely coupled system for $B \leq N$. The structure provides B alternate paths for communication and is therefore highly fault-tolerant. The value of B can

be fixed depending on the communication requirement of the system. Based on operation, the multiprocessor network can be divided into four categories, namely :

Centralized synchronous,

Centralized asynchronous,

Decentralized asynchronous and

Hybrid.

Centralized means that there is a central controller to oversee the operation of the IN and synchronous means the IN operates based on a clock cycle so that all the processors submit their requests at the beginning of the clock. The data transfer is also completed by the end of the cycle. A hybrid network may consist of a combination of the above three types. In this paper we will point out some of the previous work done in the performance analysis of various multiple-bus INs and mention the future scope of study.

A lot of research has gone into the performance analysis of centralized tightly coupled multiple-bus INs [2-4], assuming that a processor sends a request to any memory module with equal probability. Analysis has also been reported when a processor has a favorite memory [5]. As reported in the above papers, the BW linearly increases with the increase in number of buses until it is saturated by the memory access conflicts. In centralized asynchronous tightly coupled operation, the processors send the memory request packets to the central controller. The central controller builds a queue for each memory and allocates buses to these memories in a cyclic fashion. Analysis of this system has been reported in [6,7]. In a decentralized asynchronous operation, we need to follow some local area network protocols to capture the bus. Decentralized protocols adopted in token, slotted or Ethernet [8] buses can be applicable. As can be seen from the current university and industry supercomputer projects, the trend is to employ packet switching in the INs instead of circuit switching. We are in the process of analysis and design of suitable multiple-bus protocols for tightly coupled multiprocessors. No work has been reported yet

on hybrid networks suitable for processor - memory interconnection.

Analyses have been reported for all the networks in a loosely coupled environment. A loosely coupled system is similar in operation to a local area network except that the interaction in the former is at a much lower level and has very fine granularity. Analysis of a loosely coupled system is much simpler because there is no memory access conflict and the processor does not have to wait until the current request is satisfied. Because the interaction is less frequent, techniques such as context switching can be applied in a loosely coupled multiprocessor. It is usually assumed that a processor generates requests as per Poisson process and the network efficiency is measured in terms of delay-throughput characteristic. Analysis of a synchronous/asynchronous multiple-bus network is equivalent to M/G/m queueing analysis that can be obtained from any standard book [9]. A decentralized asynchronous operation is similar to the operation of multichannel local area networks whose performance have started appearing recently. Multichannel protocols and analyses for carrier sense multiple access / collision detection (CSMA-CD) have been reported [10,11]. In our opinion, these protocols are also well suited for multiprocessor operations. We have also evaluated the performance of some hybrid networks that consists of a few CSMA/CD buses, centralized and token buses [12,13]. The idea is to divert the traffic from the contention buses when the level of contention is high. As a result we obtain much better performance.

Finally we have thoroughly evaluated the reliability and availability and our results indicating that the multiple-bus performs the best [5,14]. In conclusion, we believe that multiple-bus offers a simple, flexible, fault-tolerant and easily expandable interconnection scheme. Industries always prefer a shared-bus connection [15]. If communication is a problem, let us put another bus in parallel and see what happens!

References

- [1] T. Y. Feng, "A Survey of Interconnection Networks", IEEE Computer, Dec. 1981, pp. 12-27.
- [2] L. N. Bhuyan, "A Combinatorial Analysis of Multibus Multiprocessors," Proc. Int. Conf. on Parallel Processing, Aug. 1984, pp. 225-227.
- [3] T. Mudge et. al., "Analysis of Multiple-bus Interconnection Networks," Proc. Int. Conf. on Parallel Processing, Aug. 1984, pp. 228-232.
- [4] T. Lang et. al., "Bandwidth of Crossbar and Multibus connections for Multiprocessors," IEEE Tran. on Computers, Dec. 1982, pp.1227-1234.
- [5] C. R. Das and L. N. Bhuyan, "Bandwidth Availability of Multiple-bus Multiprocessors," IEEE Tran. on Computers. Special Issue on Parallel Processing, Oct. 1985, pp. 918-926.
- [6] Q. Yang, "Communication Performance in Multiple-bus Systems", M.A.Sc. Thesis, Dept. of Ele. Eng. University of Toronto, 1985.
- [7] D. Towsley, "Approximate Models of Multiple Bus Multiprocessor Systems", IEEE Tran. on Comp., Vol. C-35, No. 3, March 1986.
- [8] A. S. Tanenbaum, *Computer Networks* Prentice-Hall, New York, 1981.
- [9] S. S. Lavenberg and C. H. Sauer, "Analytical Results for Queueing Models," Computer Performance Modelling Handbook, S. S. Levenberg, Ed. New York: Academic, 1983, pp.55-172.
- [10] M. A. Marsan and D. Roffinela, "Multichannel Local Area Network Protocols", IEEE Sel. Areas Comm., pp. 885-897, Nov. 1983.
- [11] H. Okada, et. al., "Multichannel CSMA/CD Method in Broadband-Bus Local area Networks", IEEE CH2064-4, pp. 19.1.1-19.1.6, 1984.
- [12] G. S. Selvam, "Multiple Bus Local Area Networks", M. S. Thesis, The Center for Advanced Computer Studies, Univ. of Southwestern Louisiana, 1986.
- [13] R. H. Sy, "Performance Analysis of Hybrid Local Area Network Architecture", M.S. Thesis in Preparation, The Center for Advanced Computer Studies, University of Southwestern Louisiana.
- [14] C. R. Das and L. N. Bhuyan, "Reliability Simulation of Multiprocessor Systems", Proc. Int. Conf. on Parallel Processing, pp. 591-598, 1985.
- [15] Multimax Technical Summary, Encore Computer Corporation, 1986.

Session 6: Reconfiguration Strategies

Chairperson: Sudhir Ahuja
AT&T Bell Lab.

AUTOMATICALLY RECONFIGURABLE COMPUTER ARCHITECTURE

Dr. F. Gail Gray

**Department of Electrical Engineering
Virginia Polytechnic Institute and State University
Blacksburg, Va. 24061**

**This work was supported in part by
Army Research Office Grant DAAG29-82-F-0102.**

AUTOMATICALLY RECONFIGURABLE COMPUTER ARCHITECTURE

1.0 Automatically Reconfigurable Computer Architecture

This paper describes a proposed automatically reconfigurable cellular architecture. The unique feature of this architecture is that the reconfiguration control is distributed within the system. There is no need for global broadcasting of switch settings. This reduces the interconnection complexity and the length of data paths. The system can reconfigure at the request of the applications software or in response to detected faults. This architecture supports fault tolerant applications since the reconfiguration can be self-triggered from within. The complete reconfiguration process can proceed without external interference.

1.1 Introduction

Recent advances in VLSI technology has made it possible to interconnect many small computers together to achieve high parallelism. There are many topologies for interconnecting processors. The optimum choice for interconnection strategy is frequently application dependent; therefore, most of these architectures can be used only for a small number of applications. In order to support a wide range of applications, it is highly desirable to design and build a general purpose reconfigurable multiprocessor system.

The demand for very high performance computing systems has forced researchers to consider non-traditional architectures, notably distributed/parallel systems[10-14]. This is because the exponential execution time required by many algorithms can be significantly reduced by exploiting inherent parallelism. Most of these new architectures attempt to match the underlying hardware to specific problems(algorithms) for fast, efficient execution.

Kung [1] has proposed such a class of architectures known as systolic arrays that can be used to perform a variety of highly parallel computations such as matrix multiplication, fast Fourier transformation, etc. Each processor in these arrays performs a simple and short computation and regularly pumps data in and out. But only a limited number of functions can be performed with each type of interconnection network. What is needed is a general purpose reconfigurable architecture that allows many of the special purpose architectures to be embedded in a single structure.

Considerable attention is being given to such a general purpose reconfigurable architecture in recent literature. Snyder [2] demonstrated the feasibility of such an architecture with the CHIP computer (Configurable, Highly Parallel computer), which provides a programmable interconnection structure integrated with processing elements. It is designed to provide the flexibility needed to compose general solutions while retaining the benefits of uniformity and locality that the algorithmically specialized processors exploit. It consists of a switch lattice, in which the switches are set to create the best interconnection network for the function to be executed. An external controller broadcasts a command to all the switches to invoke the appropriate architecture.

This approach has two major disadvantages. First, the setting of the switches is controlled by an external processor. This necessitates some type of global connection to all switches. Secondly, the master control circuitry becomes a single point failure site, since it would be necessary for the master control to work correctly in order for the appropriate switch setting to be invoked. Thus the failure of the master control will significantly degrade the system reliability. This is highly undesirable for failure critical applications such as automatic landing of commercial aircraft, control

systems for nuclear reactors, life support systems for medical applications, etc. Even in less critical applications, system downtime is often very expensive.

In this paper, we propose a reconfigurable cellular architecture in which the reconfiguration mechanism is distributed throughout the array instead of being a single-point failure problem. In addition, reconfiguration does not require an external processor to compute the new interconnection pattern.

1.2 Proposed Structure

Our desire is to be able to implement a set of specific architectures designed to solve a specified set of parallel computations in a single reconfigurable system. Cellular arrays are a viable computational architecture for such an implementation. A cellular (iterative) array is a collection of identical cells that are interconnected in a uniform, or regular, fashion. A cellular array is proposed for the following reasons. First, they are of highly parallel nature. Secondly, since all the cells in the array are identical the architecture becomes easily expandable without changing the current hardware in any significant way. Lastly, each processor is connected to other processors according to a regular interconnection pattern. By using regular local interconnection patterns we can avoid the use of global connections, so that the interconnection complexity will not increase with the size of the system. By making the control distributed throughout the array, the "hard core" component will be minimal. Being a planar, regular structure, such a parallel processor is well suited for VLSI implementation.

The proposed cellular structure is composed of two cellular arrays that are interconnected as shown in Figure 1 on page 3. The cellular structure consists of a "control hyperplane" and a "computation hyperplane", where for each cell in the computation hyperplane there is an associated cell in the control hyperplane. Each cell in the computation hyperplane can be either a switch or a processing element, as shown in Figure 2 on page 4. If it is a processing element it must be complex enough to realize the functions required by each of the algorithms, i.e., each processing element is some type of universal logic module, or microprocessor, that can perform a list of functions.

Each cell in the computation plane is controlled by the state of the corresponding cell in the control plane. If the cell in the computation plane is a processing element, then the control cell specifies a particular algorithm from a set of possible algorithms that the processing element can implement. If the cell in the computation plane is a switching element, then the control cell will specify a particular local interconnection of the switching element to its neighbors. The overall function to be performed by the cellular structure is defined by the global pattern of control states.

To create the desired configuration for a particular computation, one cell in the array is initialized to a "seed" state which defines the global computational task to be performed by the array. The cellular array will then utilize that information to "grow" the required pattern of states in the control hyperplane. The pattern of states in the control hyperplane invokes the desired interconnection structure in the computation plane.

1.3 Theory of Cellular Reconfiguration

The control hyperplane is responsible for assigning proper functions to the cells in the computational hyperplane. Any arbitrary cell in the control hyperplane will eventually receive information about the global function to be implemented from the "seed" state initially planted at an arbitrary location. The way in which the given information is distributed throughout the control plane to produce a desired final pattern is explained in this section. This process will be referred to as 'growth'.

The control hyperplane is an array of identical cells interconnected in a uniform fashion, where each such cell can receive state information only from its neighbors. The cell which initially receives information about the global function will communicate with its neighbors and gradually spread the information through the array to create the final desired pattern. This section explains the "growth"

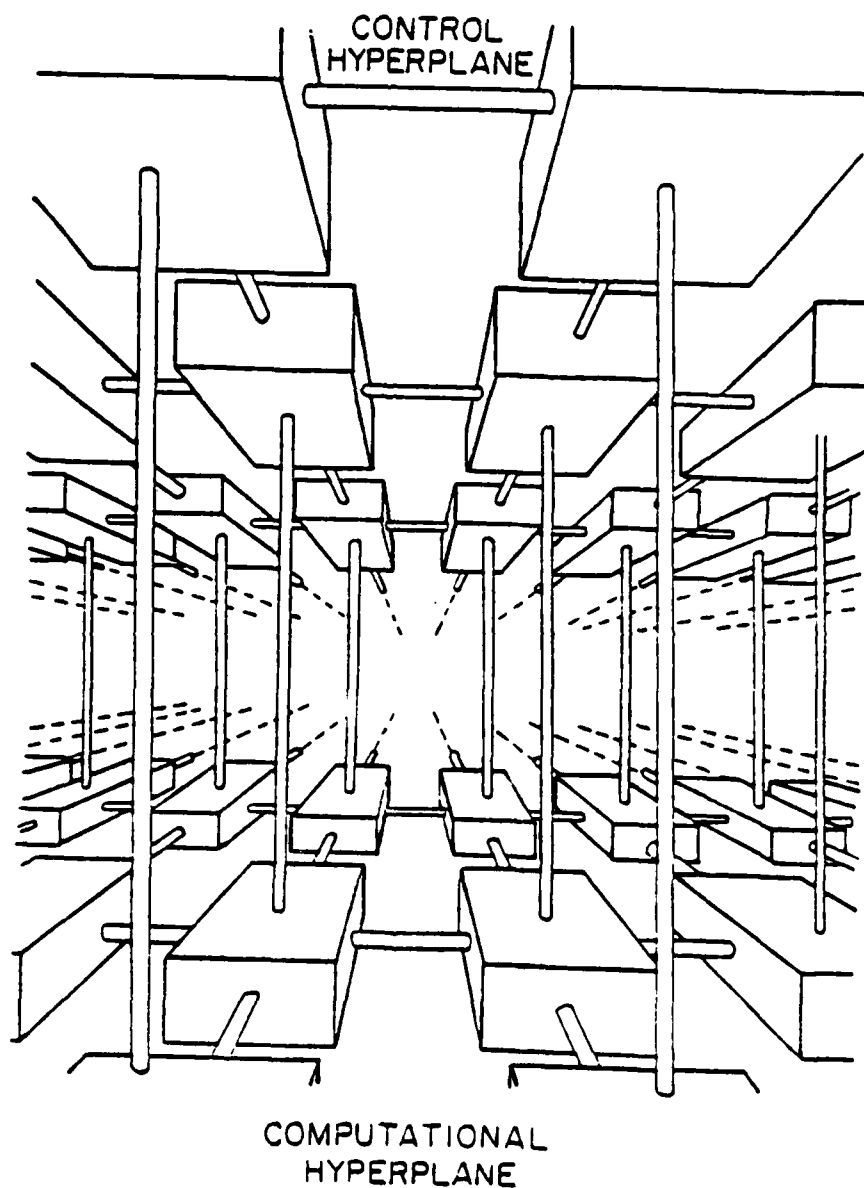


Figure 1. Cellular Architecture

Automatically Reconfigurable Computer Architecture

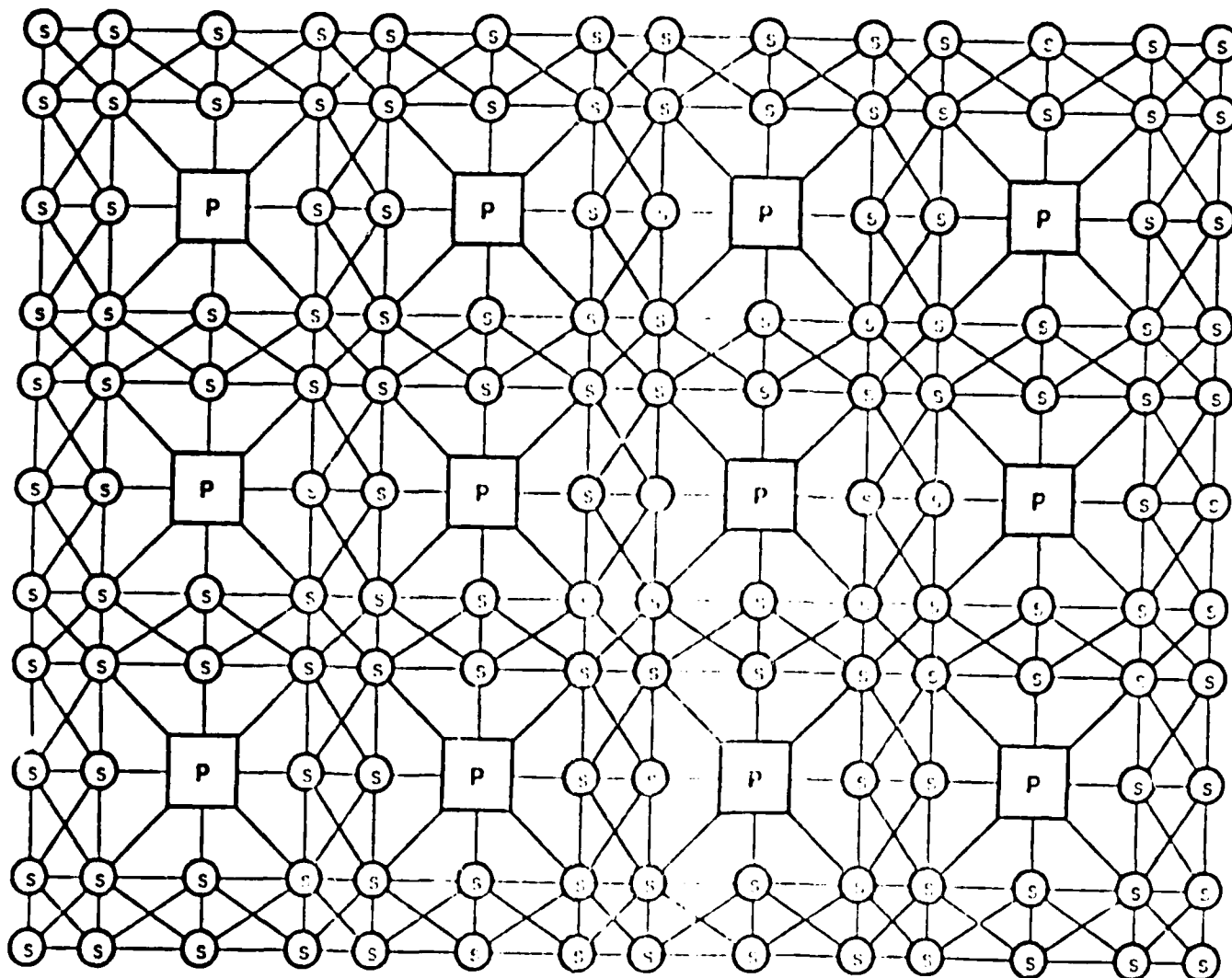


Figure 2. Computational Plane

Automatically Reconfigurable Computer Architecture

process, the way in which the given information is transmitted throughout the control plane to produce a desired final pattern. Since the control hyperplane is an array of identical cells interconnected in a uniform fashion, each cell can receive state information only from its neighbors.

Cell	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	time
...	0	0	0	0	0	0	0	S	0	0	0	0	0	0	0	0
...	0	0	0	0	0	0	0	A	A	A	0	0	0	0	0	1
...	0	0	0	0	0	0	0	B	B	C	B	B	0	0	0	2
...	0	0	0	0	E	E	I	I	E	E	E	0	0	0	0	3
...	0	0	0	C	O	M	P	U	T	E	R	S	0	0	0	4

This process is illustrated in the following example.

Each row describes the state of the one-dimensional array at a particular time. At $t = 0$, the "seed" state "S" is planted at an appropriate place in the array with all other cells in the "quiescent" state "0". At each time step, each cell observes the state of each of its immediate neighbors and its own state, then decides what its next state will be.

For example, at time $t = 0$, cell 0 observes that the cell to its immediate left is in the "0" state, cell 0 is in the "S" state and the cell to its immediate right is in the "0" state. The *local pattern* for cell 0 at time 0 is said to be 0S0. When any cell in the array is in a local pattern of 0S0 at time t , it will change its own state to state A at time $t + 1$. Similarly, if a cell is in a local pattern of 00S or S00 at time t it will also change its own state to state A at time $t + 1$. Any cell in local pattern 000 at time t will stay in local state 0 at time $t + 1$. Therefore, the result of each cell at time $t = 0$ observing its local pattern and changing to the appropriate next state will transform the *global pattern*

0 0 0 0 0 0 0 S 0 0 0 0 0 0 0

at time $t = 0$, into the global pattern

... 0 0 0 0 0 0 A A A 0 0 0 0 0 0 ...

at time $t = 1$. In like manner, the global pattern at $t = 1$ will be transformed into the global pattern

... 0 0 0 0 0 B B C B B 0 0 0 0 0 ...

at $t = 2$ by each cell applying the following local transformation

local pattern	next state
0 0 0	0
0 0 A	B
0 A A	B
A A A	C
A A 0	B
A 0 0	B

Eventually, the desired global pattern "COMPUTERS" will be reached and will remain stable if each cell applies the following local transformation.

O	O	C	O
O	C	O	C
C	O	M	O
O	M	P	M
M	P	U	P
P	U	T	U
U	T	E	T
T	E	R	E

In general, any such control hyperplane can be characterized as a tessellation automaton [3].

The *tessellation automaton* (TA) is a four tuple

$$TA=(A,Ed,X,\sigma)$$

where,

1. A is a finite non-empty set called the *state alphabet*. For our previous example, $A = \{O,S,A,B,C,I,E,O,M,P,U,T,R\}$
2. Ed is the set of all d-tuples of integers called the *tessellation space*. Here the tessellation space is said to be "d" dimensional. In our example, Ed is simply the set of all integers

... -3, -2, -1, 0, 1, 2, 3, ...

Ed defines the spatial location of each cell in the array.

3. X is an n-tuple of distinct d-tuples of integers called the *neighborhood index*. Each cell is said to have n neighbors and n is called the *neighborhood scope*. In the example, $n=3$, and $X = (-1, 0, 1)$. X defines the relative coordinates of a cell's neighbors. For example, cell 5 has neighbors (4,5,6) obtained by adding each coordinate of the neighborhood index to the cell location (5).
4. σ is a mapping from A^n into A called the *local transformation*.

Each cell will decide its next state by observing the present state of its neighbors. In the example, $\sigma(BCB) = I$, $\sigma(MPU) = P$, etc. It is desirable from the stability point of view that each cell be its own neighbor [4].

For our work we can conveniently characterize the control hyperplane as a two dimensional tessellation automaton, since the computation hyperplane is a two dimensional array of switches and processing elements. To completely describe the control hyperplane, we need to specify the neighborhood index and the local transformation σ . The only difference from the example is that the neighborhood index and the local transformation will be two dimensional in nature.

1.4 Summary of Results

This presentation describes the results of research into the topic. No attempt will be made to describe the results in great detail.

1.4.1 Design Example

A complete design example that incorporates four representative architectures has been developed [17]. These were chosen to achieve a variety of research architectures. Included were banyan networks [Gork76], a fault tolerant architecture [Prad82], the hypertree [Good81], and the lens strategy [Fink81]. The results show that the complexity of the design is practical for present VLSI technology.

Automatically Reconfigurable Computer Architecture

1.4.2 I/O Algorithm

Since the structure of the array changes from architecture to architecture, and since the location of the active cells may change dynamically due to faults, it is necessary to provide a dynamic connection from the I/O ports of the array to the active region. This dynamic path allocation must utilize only good cells. A complete specification for an I/O algorithm has been developed.

1.4.3 Determination of Fault Free Spaces

A distributed algorithm has been developed to determine the sizes of maximal regions of fault-free cells in the array. Faulty cells are surrounded by a quarantine wall. Each cell in the array passes a continually updated version of a parameter called the s-value to each of its neighbors. When the s-value distribution becomes stable, each cell in the fault-free region will know exactly how many fault-free cells are available in all four directions from its location. This distributed space finding algorithm plays a crucial role in the development of distributed fault-tolerant reconfiguration algorithms for the proposed structure.

1.4.4 Fault Tolerant One-Dimensional Architecture

A complete set of algorithms, including the required transformations, has been developed for both fault-diagnosis and reconfiguration in a one-dimensional architecture [16]. Both single and multiple faults are covered. Distributed control is maintained for all procedures. Results regarding the rate of growth of patterns and the number of time steps needed for reconfiguration have been derived.

1.4.5 Multi-Dimensional Architecture

A complete set of algorithms for fault diagnosis and reconfiguration in multi-dimensional architectures has been developed. Although applicable to architectures of any dimension, two dimensional examples are emphasized.

1.5 Future Directions

This section identifies problems that need to be solved to make the proposed architecture more efficient and to increase the range of applications.

Previous work has focused on the use of the array for non-real time applications. High availability was the goal. There are no provisions for recovery during a computation. The system can reconfigure but then the computation in progress must be restarted from the beginning. For real time applications, there must be some means for saving data during reconfiguration. Such applications require no gaps in data collection. A possible solution is to utilize a data checkpointing system, with queues to hold data while reconfiguration is in progress. The problem of transferring the state of the failed processor to its replacement is also a difficult problem. One possible solution is to utilize shadow processors, so that there are always extra processors running each task in lock step. In this way, it might be possible to quickly switch from the faulty processor to a good one. Although these problems have been studied in other environments, applying them to an array processor with embedded control will require considerable effort.

Although the problem for reconfiguration in the presence of faults has been extensively investigated, no method for fault detection has been formalized. Several possibilities exist. If a processor could execute a self test, or if neighboring processors could test a processor, then no additional hardware would be required. However, this approach would require extensive software, and would require considerable time. Faults could be present for some time before detection. Some real time applications could not tolerate this amount of fault latency. Another possible solution is to use redundant processors, each performing the same computation in lock step. The outputs could be compared, data errors would be detected with no delay, and system reconfigura-

tion could immediately be invoked. The advantages of this approach are less software overhead for testing, and low fault latency. The disadvantage is considerable hardware overhead. Since hardware costs are going down while software costs are going up, perhaps this last alternative might turn out to be best in the long run.

Time for reconfiguration is also a potential problem in real time systems. The current reconfiguration algorithms proceed by clearing out the entire array when a fault is detected. Regrowth of the desired control pattern then proceeds with faulty cells quarantined. Although this process is acceptable for non-real time applications where the computation can be restarted from the beginning once the reconfiguration is complete, real time applications require fast efficient processing. For this reason, partial clearing and partial reconfiguration will probably be necessary. This will require considerable modification of the reconfiguration algorithms.

A final concern is efficient use of available resources. The present algorithms do not allow faulty cells within the active processing array. Modification of the control algorithms to allow this condition would allow more efficient use of facilities.

1.6 References

1. Kung, H.T., "Why Systolic Architectures?" *Computer*, January 1982, pp 37-46.
2. Snyder, L., "Introduction to the Configurable, Highly Parallel Computer", *Computer*, January 1982 pp 47-56.
3. Yamada, H., and Amoroso, S., "Tessellation Automata", *Information and Control*, 1969.
4. Walters, S.M., *Pattern Synthesis and Perturbation in Tessellation Automata*, Ph.D. Dissertation, Virginia Tech., Jan 1980.
5. Von Neumann, J., *Theory of Self-Reproducing Automata* University of Illinois Press, Urbana, Illinois, 1966.
6. Good, I.J., "Normal Recurring Decimals" *J. London Math. Soc.*, Vol.21 PP 167-169, 1946.
7. Goke, L.R., *Banyan Networks for Partitioning Multiprocessor Systems*, Ph.D. Dissertation, University of Florida, 1976.
8. Goodman, J.R., and Sequin, C.H., "Hypertree: A Multiprocessor Interconnection Topology", *IEEE Transactions on Computers*, Vol c-30, December 1981.
9. Finkel, R.A., and Solomon, M.H., "The Lens Interconnection Strategy", *IEEE Transactions on Computers*, December 1981.
10. F.G. Gray and R.A. Thompson, "Reconfiguration for Repair in a Class of Universal Logic Modules", *IEEE Transactions on Computers*, Vol. C-23, November 1974, pp. 1185-1194.
11. B.A. Prasad and F.G. Gray, "Multiple Fault Detection in Arrays of Combinational Cells", *IEEE Transactions on Computers*, Vol. C-24, August 1975, pp. 794-802.
12. F.G. Gray and R.A. Thompson, "Fault Detection in Bilateral Arrays of Combinational Cells", *IEEE Transactions on Computers*, Vol. C-27, December 1978, pp.1206-1213.
13. J.R. Armstrong and F.G. Gray, "Fault Diagnosis in a Boolean n-Cube Array of Microprocessors", *IEEE Transactions on Computers*, Vol. C-30, August 1981, pp. 587-590.
14. S.M. Walters, F.G. Gray, and R.A. Thompson, "Self-Diagnosing Cellular Implementations of Finite-State Machines", *IEEE Transactions on Computers*, Vol. C-30, December 1981, pp. 953-959.
15. F.G. Gray, "General Purpose Reconfigurable Architecture", *Proceedings of the 1982 International Conference on Circuits and Computers*, New York, NY, September 28-October 1, 1982, pp. 122-125.

16. R. Kumar and F.G. Gray, "A Fault Tolerant One-Dimensional Cellular Structure", *The 4th International Conference on Distributed Computing Systems*, May 14-18, 1984, San Francisco, CA, pp. 472-483.
17. N. Gollakota and F.G. Gray, "Reconfigurable Cellular Architecture", *1984 International Conference on Parallel Processing* August 21-24, 1984, Bellaire, MI, pp. 377-379.

Implementation of a Buddy Failure Recovery Concept in the NOVAC System

B.C. Desai
Department of Computer Science
Concordia University
Montreal, Quebec. H4B 1R6. Canada

ABSTRACT

In exact computation, a number of problems exist, the solution to which demands an exhaustive search and hence a great deal of computing time. The algorithms used are simple but the computation involved is so great that it cannot be done economically on a large scale time-shared general purpose computer. The NOVAC project at Concordia consists of a dynamically variable tree structured system for solving a class of combinatorial problems. The proposed multiprocessor structure consists of loosely coupled processors with no shared memory; each processor in the system can be a master or a slave or both. This system is to be built with off-the-shelf mini and micro computers, and interconnected using an inexpensive asynchronous bus. The logical structure, consisting of a hierarchy of masters, each with a number of slaves is natural for the set of problems which can be split-up into a number of identical sub-problems. The time required to solve such a problem could be, depending on it's size and nature hours, days or weeks. The high costs involved in having an absolutely reliable system for such duration necessitated a recovery system using the buddy concept. Herein a failure of one of the processor of a pair of buddy processors could be tolerated. The task assigned to the failed processor could be picked up from the last check point state stored in an on-line auxiliary storage device.

1. Introduction

There are many problems in exact computation, requiring a great amount of computing time to solve them. The computations involved are simple but the amount of computation is so great that it cannot be done economically on a large scale general purpose computer. A number of systems [1, 2] have been proposed to exploit the parallelism in such problems. A computer structure in the form of a tree has been proposed in [3] to solve problems that may be expressed with recursive algorithms. A number of adaptive computer architectures have been proposed previously [4, 5] for the application of such problems.

However, many of these systems are in the developmental or experimental stage and are not very extensive in terms of general availability. The development in VLSI technology has made it

possible to produce inexpensive micro and supermicro processors with a fairly significant amount of computing power. Distributed systems which evolved as a result of this situation, consist of multiple computers and devices interconnected, using a network to form a coordinated system. Such a distributed system offer a novel approach to implement fault-tolerant and highly available computing [8-11]. The NOVAC project [6] at Concordia consists of a loosely coupled, Non-tree structured multiprocessor system which can be dynamically structured into a Variable tree to solve a class of Combinatorial problems. The structure, consisting of a hierarchy of master, each with a number of slaves is natural for the set of tasks which can be split-up into a number of identical sub-tasks. In a multi-level system a slave can further partition its subtask and pass the sub-subtasks onto its own slaves. A prototype of this system is built with off-the-shelf mini-computer (PDP11/34) and micro-computers (LSI/73) and interconnected using inexpensive asynchronous bus.

Our design (figure 1)proposes a set of processors logically connected as a "tree" and satisfying the following conditions.

- (i) Each processor is either a master, a slave, or both.
- (ii) No memory is shared between processors.
- (iii) A master can instruct its own slaves to perform tasks, and subsequently obtain results from the slaves; no other communication is possible.

The capacity of the communication channels is not critical for this project. We anticipate that in a typical problem, a master will instruct a slave by transferring a few thousand bytes of code and data to it, and the slave will then run for at least several minutes before returning a solution. This is actually the most pessimistic scenario, because if the problem is very large, the amount of data transmitted to the slave will be about the same, but the slave may execute for hours or days before replying.

2. Reliability.

Reliability is always an important consideration in the design of a system which has to perform correctly in a critical environment. Physical devices have an inherent failure rate and as such components of computer systems will fail. The components in the current computer systems are built using technologies that are more reliable than earlier technology. However, the probability of component and subsystem failure in a complex system is never reducible to zero. Systems for real-time processes, such as air traffic control, defence and space systems, usually incorporate various mechanisms which can detect faults and take appropriate corrective actions to make the system available with a very high probability for the life of the mission. There are essentially two methods to increase the reliability of a computing system [7]: there being the fault

avoidance (fault-intolerance) and the fault tolerance. In the fault avoidance method, the reliability is increased by the use of conservative design guidelines, using highly reliable components that have been subjected to an initial burn-in phase etc. However, in spite of these techniques, failures will occur.

In fault tolerance, on the other hand, the objective is for the system to perform to the desired specification even in the presence of faults. The use of redundancy is the usual approach. Redundancy in a computing system could be considered to be in two dimensions, namely components or time. Component redundancy is in the form of extra components, subsystems and systems; time redundancy is usually provided by software. The proposed buddy recovery method falls in the area of fault tolerance with time redundancy.

Most of the commercially available systems [11] are made fault tolerant by providing redundant or spare processors and/or redundant communication links. When an active system component fails, its tasks are dynamically transferred to spare components, thereby allowing system operation to continue. In most of the systems it has been assumed that reconfiguration and recovery are directed by a central or global supervisor, which is often the most vulnerable part of the system.

3. Buddy Recovery Concept

In this section, we propose a failure recovery strategy for a task oriented distributed computing system, where tasks may spawn subtasks which execute in other nodes. A node is assumed to be capable of detecting failures of the processors at other nodes. The only symptom of malfunction assumed for processors is complete failure or "death". In order to make such a multiprocessor system fault tolerant, it is necessary to develop strategies which after the occurrence of a processor death will permit the remaining processors to cooperate and agree on a reconfiguration of the system. We assume that the interconnection network will not fail. Each node performs its tests independently according to some locally determined schedule.

One of the recovery strategies often used is a write-in-place scheme in which the system must save sufficient information to undo the effect of an action if it fails. Backward error recovery i.e. resetting an erroneous state of a system to a previous error free state, is an important general technique for recovery from faults in a system. Especially in systems where errors were not foreseen. However, the operation of backward recovery can be complex particularly if the implementation of the system is multi-level.

In a conventional multiprocessor, a task is re-executed after re-initialization of the multiprocessor when a permanent hardware fault is detected. This imposes a significant overhead, since the computation between the start of the task and the time of the fault is lost. If the task is distributed on different processing units in the multiprocessor then a failure in a sub-task would imply restarting the entire task, resulting in high initialization overhead.

The proposed system takes advantage of the multiple processing units to allow recovery from hardware failures. Briefly, the detection of a processor failure by another processor, prompts the replacement of the faulty processor by this processor; and resumption of its task from a non-faulty state of computation prior to the failure.

The concept of the buddy system involves pairing of two processors. A processor P_1 , which is given a task is paired with another processor P_2 ($=P_{b1}$). P_2 may also be assigned another task and paired with processor P_3 (P_{b2}). The buddy processor P_{b1} , is responsible for testing the state of the processor P_{b1} . If P_{b1} finds the processor P_1 in a failed state then it initiates the recovery procedure. The task which was being performed by the failed processor (P_1) is taken over by its buddy processor (P_{b1}). Sub-tasks initiated by failed processor, P_1 , can proceed normally without being re-initialized. Calls for the downloading of the tasks and the state save points are designed such that the buddy processor can recover the task from an advanced state of computation.

The type of failure which we are considering is the failure of a processor while it is executing a task. The basic idea of the buddy failure recovery concept is the following. For each task t_1 that is assigned to a particular processor P_1 , a buddy processor P_b is selected. The buddy processor P_b , is cognizant of the status of the processor P_1 and the task t_1 , including any of its subtasks t_{1j} . In addition the processor P_1 stores the state of the task t_1 , at regular intervals in an on-line auxiliary storage device. This device is accessible to the buddy processor P_b . In the case of a failure of P_1 this enables its buddy processor P_b to recover from the last non-faulty state stored by the failed processor rather than re-executing the task which was being executed by the processor that failed. Also the system design allows normal continuation of any sub-task t_{1j} created by the processor P_1 before its failure, and the reporting of its results are forwarded to P_b .

Selection of the buddy processor is left to the processors; even, slave processor, on receiving a task to be performed from a master processor, chooses a buddy processor based on its own capability criteria, and then informs its master processor of the identification of the buddy processor.

4. ARCHITECTURE & COMMUNICATION REQUIREMENTS

A reliable and survivable interconnection system must feature good operational security at three levels: the architectural or structural level, the physical implementation level and the communication control level.

In the proposed system tasks are executed by a set of physically distributed computing elements. These computing elements must communicate with each other in order to co-ordinate their actions so as to achieve the global system objectives. Each processor has its own private main memory, and on-line secondary storage. The latter is not only connected to the processor but also to the global bus by its device controller. A processor, as long as it is 'alive' has exclusive access to the on-line storage. On the failure of the processor the controller allows other processors to access the on-line storage in a read only mode.

The global bus structure (Figure 1) allows direct communication between processors and maps the logical master/slave structure. Connecting the on-line storage to the bus allows the buddy processor (P_{B1}) to read the last state saved by the processor (P_1) before its failure. Using this methodology reduces the load on communication network since it does not require every state save to be sent to the buddy processor.

- (i) Direct communication is possible between any two processors.
- (ii) Failure of any one or more of the processors does not interfere in any way with the communication among the remaining processors.
- (iii) In the case of a processor failure, the on-line auxiliary storage of the failed processor is accessible to other processors for read purposes.

5. IMPLEMENTATION

The user program is loaded onto the root processor, which assumes the role of the overall master. The master processor sub-divides the given task into sub-tasks and sub-sub-tasks according to the users directives as specified in the source program. The master processor also generates the machine code of each individual task i.e. for each sub-task in the system. It also associates a unique identifier with each sub-task. The first step is broadcasting of all sub-tasks along with their unique identifier by the master processor. This broadcast is picked up by other processors in the network and stored on their respective on-line secondary storage. This one time initial broadcast and storage of sub-tasks enables the system in reducing the load on

the communication network during the execution stage since it would not be necessary for a processor to pass the code of the sub-task to a slave processor. During the execution period, whenever a processor needs another processor to act as its slave processor and execute a sub-task, it broadcasts a request for a slave over the bus, and an idle processor responds by transmitting its id. The master then passes onto the slave processor the following information: the processor's (master) ID, the ID of its buddy, the task ID and the associated parameters.

The slave processor chooses a buddy processor and sends the buddy processor's ID to the master processor, which is also received by the master processor's buddy.

As a result of the above specified sequence, the slave processor knows the identifications of its buddy processor, of its master processor and of the buddy processor of its master processor.

Whenever any processor creates a slave to execute a sub-task, it informs its buddy. This will keep the buddy processor cognizant of all the tasks down loaded by the processor; the ID of the slave and the slave's buddy. In case of a processor failure, this is needed by the buddy processor since it would be receiving the results from the tasks being performed by the slave processors of the failed processor.

On completion of the task the slave processor needs to report the results back to its master processor; if it does not receive an acknowledgement back from the master after a prespecified time it sends the results to the master processor's buddy.

Recovery in case of the failure of a processor is initiated by its buddy processor. When a processor detects the failure of its buddy it may be in either of two states - idle or busy. If it is in an idle state then it starts processing the buddies task otherwise it first finishes the task it is performing and then takes up the failed processors task.

6. Conclusion

In the buddy recovery scheme, a processor assigned a task depends on its buddy processor for the completion of the task at hand in case of a failure. The processor communicates periodically with the buddy processor to indicate that it is 'alive'. Failure of this communication initiates recovery procedure in the buddy to retrieve the last correct state of the task and the task proceeds from that point. No recomputation of the task (and its sub-tasks) are not required.

The scheme is designed for a failure in the processor unit and not in the communication facilities or the secondary storage.

device. Schemes used in commercial fault tolerant systems as dual-bus, dual port device controller and dual-ported devices are too expensive for a computation intensive environment requiring hundreds of processors.

The implementation of the recovery scheme is in progress on a prototype.

References

- [1] B.C. Desai, "A Parallel Processing System to Solve 0-1 Programming Problem", Ph.D. Thesis, McGill University, January 1977.
- [2] W.A. Wulf and C.G. Bell, "Comp - A Multi-Mini-Processor", Proceedings AFIPS 1972 FJCC (4), AFIPS Press, 765-77.
- [3] B. Buchberger, J. Fergel and F. Lichtenberger, "Computer Trees: A Multicomputer Concept for Special Purpose Parallel Processing", Microprocessors and Microsystems, 3,(6) 1979.
- [4] S. Sacharen, B.C. Desai, E. Cerny, "Multiple Bus Interface", DECUS Canada, Spring 1978, Ottawa, Ontario, Canada.
- [5] S.I. Kartashev and S.P. Kartashev, "Multicomputer Systems with Dynamic Architecture", IEEE Transactions on Computers, 28,(10) 1979, 704-21.
- [6] B.C. Desai, J. Opatrny, C. Lam, P. Grogono, J.W. Atwood, N. Cabilio, "NOVAC - A Non-Tree Variable Tree for Combinatorial Computing", Proc. of the 1982 International Conference on Parallel Processing. IEEE, pp.193-199.
- [7] D.P. Siewiorek, "Architecture of Fault-Tolerant Computers", Computer, August 1984, pp.9-18.
- [8] A. Avizienis et al., "The Star (Self-Testing and Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design", IEEE TC-20-11, pp.1312-1321.
- [9] D.P. Siewiorek et al., "A Case Study of C.mmp, Cm and C.vmp: Part 1 - Experiences with Fault Tolerance in Multiprocessors Systems", Proc. IEEE. 66-10, pp.1178-1199.
- [10] D.A. Rennels, "Distributed Fault-Tolerant Computer Systems", Computers, March 80, pp.55-65.
- [11] G. Sedina, "Fault-Tolerant Systems in Commercial Applications", Computer, August 1984, pp.17-19.

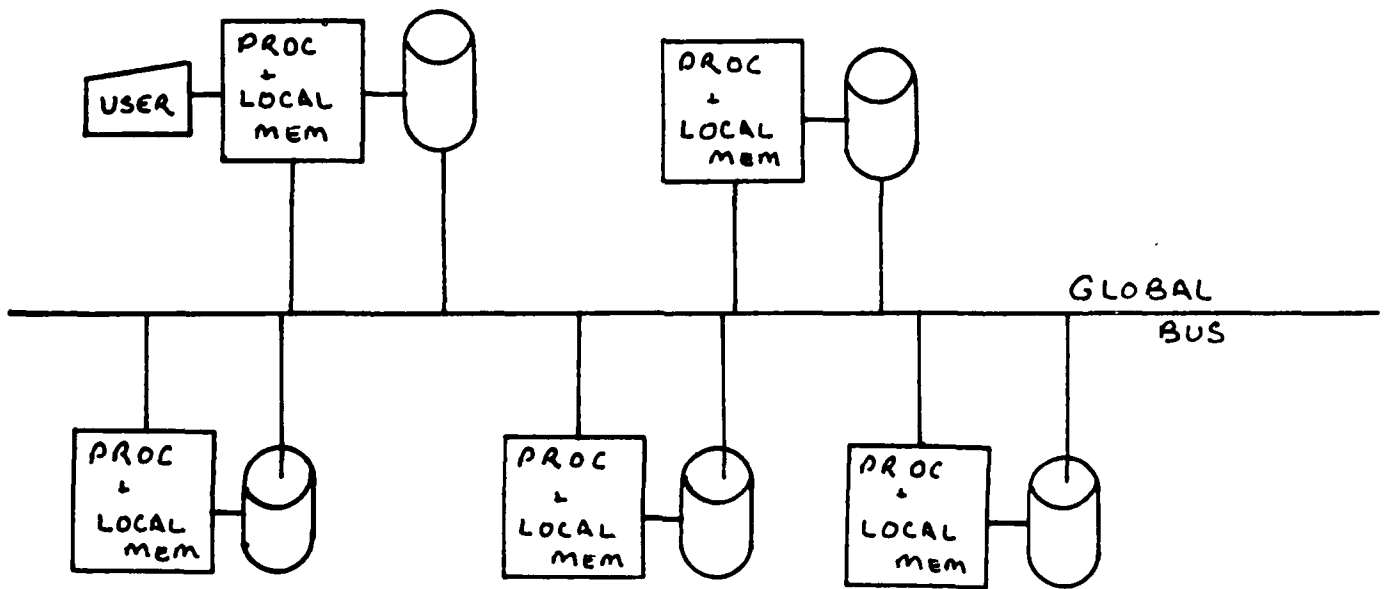


Figure 1. Proposed System

Clouds: A Support Architecture for Fault Tolerant, Distributed Systems.[†]

P. Dasgupta and R. J. LeBlanc Jr.
School of Information and Computer Science
Georgia Tech, Atlanta GA 30332

1. Introduction

The *Clouds* project at Georgia Tech was initiated to conduct research into failure resistant, efficient distributed architectures and operating systems. The project used state of the art techniques to design a distributed operating system kernel that can be supported on conventional, unreliable hardware, and be more reliable than the underlying electronics. Several approaches to the problem were considered, and after substantial research and construction effort, the current design emerged.

This paper presents an overview of the *Clouds* operating system. *Clouds* is a distributed operating system providing support for fault tolerance, location independence, reconfiguration and transactions. The implementation paradigm uses objects and nested actions as building blocks. The paper also discusses subsystems and applications that can be supported by *Clouds* to further enhance the performance and utility of the system.

2. The Clouds Project

In recent years there has been considerable interest in developing distributed computing systems. Distribution of computing resources suggests many possible benefits including greater flexibility, enhanced computing power through greater parallelism and application of more resources, and increased reliability. But the promise of distributed systems have been tarnished by a large set of problems that accompany such systems. The *Clouds* project was initiated with a determination to overcome these problems through clear, simple, elegant design, that would lead to a integrated, useful distributed system.

2.1. Toward Object Oriented Systems Design

The approach currently advocated by the research community is an object based architecture (see section 2.3 and 2.4). Every system (and user) function on an object based system is encapsulated in objects. The data these operations access are also encapsulated in the same object as the operation. These objects form units of synchronizable shared data structures and units of recoverable data in case of crashes.

The conceptual mechanism to access objects are procedure calls. However, most conventional systems do not have the support to allow *remote procedure call* (RPC) mechanisms needed to support object invocations. Thus a solution to implementing objects on conventional operating systems is to use the *active* object paradigm. In this technique, a process is associated with every object, and receives requests (via messages) from other process wishing to access the object. The object handling process is complex, with abilities of interleaved execution, non-blocking receives and some parallelism (as it has to handle several clients at the same time).

This is the approach adopted in the the Eden system at Washington [Alme83], the CRONUS system at BBN [Schn85] Argus at M.I.T. [Lisk83a] and several other systems. Although it is better at failure tolerance than a message based system, this approach still has its pitfalls, such as the way the object manager has to be implemented. It also uses a large number of processes (one per object), and has complex process handling code.

Support for distributed object based systems is built into the Argus language that is supported on the Argus system. Argus uses guardians and objects to create conceptual nodes. However, most of the object calls in Argus are handled through messages and slaves [WeLi83].

Clouds uses the object based approach with passive objects. The invoking process carries its thread of execution into the object. Further, *Clouds* does not stop at objects being the sole units of recovery. Object are units of recoverable data, *actions* are units of recoverable active components. The *Clouds* paradigm of using actions and objects for a reliable distributed system has a potential for large payoffs due to its simplicity, ease of implementation and efficiency.

2.2. The *Clouds* Architecture

The architecture of a distributed system can be partitioned into two main areas: The hardware configuration and the operating system structure. *Clouds* is designed to run on a set of processors, loosely coupled over a medium-to-high-speed network. The prototype configuration is shown in Fig. 1.

The prototype consists of three VAX/750 computers with 3 Meg memory connected through a high speed *Cluster Interconnect* (CI). The front end network consists of an Ethernet. This is the *Clouds* gateway to the external world. (Currently, both the back end network and the front end network are on the same Ethernet.) Users access the *Clouds* system through desktop computers (IBM-PC/XT computers in our prototype) through the Ethernet. The disk drives used for secondary permanent storage are dual ported. This allows reassignment of disk drives in case of processor failure and helps in reconfiguration. The front end Ethernet also serves as a vehicle for easy transfer of user-processor assignments. In case of failures, users can be virtually transparently floated to another serviceable processor.

2.3. Objects

The operating system structure of *Clouds* is based on the action/object paradigm. All permanent system components in *Clouds* are *objects*. Objects form a clean conceptual encapsulation of data and programs. They are useful in providing synchronization and recovery as will be described later. The objects are accessed by processes on behalf of actions.

In a simplistic view, an object is an instance of an abstract data type (cosmetically similar to *modules* in Modula-2 or *classes* in Simula). The object encapsulates permanent data and a set of routines that can access (read or update) the data. The only access path to the data contained in the object is through the routines (or operations) defined in the object. To the external world, the object is thus an entity providing a set of entry points. [Jo79, Wulf74]

Clouds object are more powerful than just a black box containing some procedures and static data. It also contains a stack (temporary data), heap (permanent dynamically allocated data), and powerful support for concurrency control and recovery. An outline of a *Clouds* object is shown in Fig. 2.

2.4. Actions

Actions are partially ordered sequences of operations on objects that transform the state of the objects from one consistent state to another. Actions are used to specify units of synchronization and recovery.

A user on the *Clouds* system can start up a transaction. A user transaction is a *top-level action*. A top level action is an atomic unit of work, that either effectively terminates and causes permanent updates to recoverable objects, or does not leave a trace.

A top level action can spawn more actions or subactions. These are nested actions. They can run concurrently with other subactions and the top level action, or they can be spawned sequentially. The subactions inherit the locks, and the views of the parent action, but execute independently. The subactions may abort or commit (conditional commit), and the termination status is returned to the parent. A top level action may decide to

[†] This research is funded in part by NASA grant NAG-1-430 and by NSF grant DCR-8316590.

abort or to commit after all the nested actions have terminated.

2.5. Synchronization and Recovery

Clouds uses locks as a basic mechanism for synchronization amongst actions. (The locking mechanisms are not ingrained into the *Clouds* design; some other forms of synchronization such as timestamps can easily be substituted.) The object designer can classify an object as *synchronized* or *non-synchronized*. Synchronized objects are automatically synchronized using the 2-phase locking paradigm. The synchronization in non-synchronized objects is left to the programmer. Synchronization primitives such as locks and semaphores are provided for do-it-yourselfers. This allows semantic based synchronizations.

When an object is defined as *synchronized*, the object compiler includes default code in the object entry and exit points to adhere to the 2-phase locking protocol. Each operation is classified as read or update operation depending upon the semantics of the operation. Invoking a read operation causes the invoking action to acquire a read lock on the object (if possible, or the process waits until the lock is obtained.) Similarly invoking a write operation causes the acquisition of a write lock, or the upgrade of a pre-existing read lock to a write lock.

Locks are not released when the operation returns or terminates, but are released by the commit phase, which is also handled by the object as described below.

Objects can also be defined to be *recoverable* or *non-recoverable*. That is any modification to the object is not made permanent until the action that caused the modification successfully terminates. Recoverable objects have default operations "Commit" and "Abort" defined in them. One of these operations is invoked automatically depending upon the success or the failure of the action. Commit causes the updates to be made permanent and abort obliterates changes.

2.6. The Salient Features of *Clouds*

Distributed system architecture has been a research topic for some time now. The basic choice for a distributed environment gaining popularity is the object based approach. Coupled with actions, objects are a powerful and yet simple concept, that is well suited to handle distributed computing environments. However, not many prototypes have emerged, although some are under construction. Some of the notable ones are the TABS project at CMU [Spec84], Argus at MIT [Wei83, Lisk83a, Lisk83b], Zeus at Austin and Bloomington [Brwn83], CRONUS at BBN [Alme83], Archons at C.M.U. [Jens82] and Eden at Washington [Alme83]. The current state of the art in constructing distributed operating systems and environments is definitely in this direction.

What all these projects have in common with *Clouds* is the usage of objects and actions. The object is a powerful construct that can effectively be used to handle some of the problems of distributed environments. An object not only encapsulates data, but also the operations that can be invoked on the data. The data contained in the object can only be accessed by the operations or functions defined in the object. Thus the object can control access to the data in a fashion the programmer designed it to. Note that even though the object itself may be passive, the object can control access and enforce synchronization through semaphores or equivalent mechanisms.

Actions are used to preserve consistency and avoid partial executions. Actions also take care of the failure recovery problem inherently. Since a failed action never leaves a mark on the system, and an action can successfully commit only if all its components executed successfully, the state of the permanent storage is always consistent.

Clouds is different from the mentioned object based systems in several respects. First it implements passive objects. In the pure object based system the object is in fact a passive segment of code and data, that has no thread of execution in it unless invoked by a process. The object can support as many threads of execution inside

it as there are concurrently invoking processes (if the synchronizations rules allow.)

The basic primitive supported by an object based system is object invocation. This is the most frequent activity done by the system. Some systems overlay the object invocation mechanisms on top of a conventional operating system. In this case each object invocation has to trickle down through several layers of software in order to work. The *Clouds* approach is to build object invocation as a primitive supported at the lowest level of the operating system, namely the subkernel.

The action management is also supported at a low level in the kernel. Thus the basic function of the *Clouds* kernel are object invocation, action handling, synchronization, recovery and commit.

In *Clouds* both hardware and software failures are modeled by the same paradigm. All failures lead to basically one or more aborted actions. The fault tolerance is achieved by reconfiguring the system and restarting failed actions after the failure is detected.

Reconfiguration capabilities are another *Clouds* novelty. This comes in two flavors. Upward reconfiguration takes place when hardware entities (processors, disks, network connections) are added to the system. Downward reconfiguration is the opposite. Downward reconfiguration can be automatic in case of detected failures or forced in cases of premeditated shutdowns. The ability of the system to reconfigure itself gives rise to two significant advantages. In case of crashes it allows normal system functions to continue. Also system components can be selectively disconnected from the live system for hardware maintenance or software maintenance (replacement of kernel level code) without affecting any activity on the system.

3. Subsystems and Applications

Clouds provides a generalized support layer for many applications and subsystems that effectively utilize the power of the object based, action oriented environment. In this section we briefly discuss the implementation of a probe-based monitoring subsystem that enhances the fault tolerance of *Clouds* and a relational database that utilizes *Clouds* support for a reliable distributed data management system.

3.1. Probe based System Monitoring

The key to improved fault tolerance lies in the implementation of a mechanism for the system to monitor itself. The monitoring can be at several levels, discussed later, but the basic components of the monitoring system are *probes*.

Probes in *Clouds* are a form of emergency status enquiries, that can be sent from a process to an object or to another process. When a probe is sent to an object, the probe causes the invocation of a *probe-procedure* defined by default in the object. The probe procedure returns to the caller a status report of the object. This includes the status of the synchronization mechanisms, the actions currently executing in the object and other relevant information (Fig. 3). Probes can also be sent to processes or actions. A process does not have to explicitly receive a probe. The probe causes a process thread of control to jump to the probe handler, irrespective of the current status of the process.

The probe handler/procedures are scheduled and executed at higher priorities than the regular process scheduling priorities. Since the probes cause an immediate, asynchronous reply, and the probes are not suspended by synchronization mechanisms, the time taken by the probe to return to the sender is not dependent on unpredictable conditions, or heavy processing loads. Thus timeouts can be quite effectively used for receiving replies from probes. Thus using this scheme we can detect the difference between a dead machine and a slow one, with a high degree of accuracy. A primary/backup paradigm using probes can be implemented to form the basis for implementing fault tolerant actions.

3.2. System Health Monitoring using Probes

The probe system can be effectively used to implement a primary/backup paradigm that achieve fault tolerant actions and a system monitoring subsystem that keeps a health status of the system. The monitoring system integrates with the reconfiguration system in a manner that enhances the failure tolerance of the system. Details are omitted for the sake of brevity. The interested reader is referred to [Da86]. The monitoring system is depicted in Fig. 4.

3.3. An Object Based Database System

One of the notable differences in structure between conventional database systems and a system supported by *Clouds* is the storage mechanism. Instead of files, we have a more powerful construct namely objects. In the following sections we describe how to implement a database system, using the object paradigm. Subsequently we discuss approaches to implement concurrency control and transaction commit for the database objects and transactions under the *Clouds* environment. We also provide insights into the effective management of the distributed database and how to provide support for data replication (*Clouds* does not support replication).

Virtually any kind of database system can be supported in the object based architecture. However to avoid getting into all the design approaches for various data modeling paradigms, we choose to discuss the most popular database model, the relational database model. The approaches for implementing other models would be different, but can be derived from the basic ideas in our discussions.

3.3.1. Objects and Relations

The basic building blocks in a relational database are relations and the relational operators that access the relations. At a slightly lower level are the access mechanisms used for fast access to individual or groups of tuples in the relational tables using key searching, indexing or hashing techniques.

The straightforward way to implement an object-based relational database system is to use a relation per object scheme. An object holds all the data of the relation and contains the access mechanisms to access the data. Thus the object defines operators that do key lookups, projections, tuple insertions, tuple deletions range queries and other such operations on the objects. A good feature of this approach is that the object can be encapsulated and be independent of any systemwide definition of structure or storage mechanisms. The internal structure of the object, that is the data organization (binary tree, B-tree, table unsorted), is not visible to the database system from outside and thus different relations can be organized in different ways and yet look functionally identical. The organization of each object could be tailored to the method that suites the data contained and the size of the object. Organization of an object using this scheme is shown in Fig. 5.

There are some disadvantages to organizing the relations in this manner. we have designs that solve most of the problems. Concurrency control and recovery methods integrate well with the automatic support provided by *Clouds*. Our approach yielded some surprisingly clean interfaces to between various types of objects and the database manipulation routines. Most of the details of the design and structuring of the database system is omitted for brevity.

Using the object based approach we can determine locking granularities local to the object and use fragmentation schemes if necessary in a manner transparent to the relational access routines and database application programs. Also replication can be effectively handled. An overview of a scheme that achieves good locking granularities through fragmentation is shown in Fig. 6. For further information about the techniques, the reader is referred to [DaMo86]

3.4. Potential Research Issues

Most of the potential research areas that will lead to successful designs of distributed operating systems have been discussed above, along with suggestions of some solutions involving *Clouds* philosophies. To reiterate, the important aspects that need further research and could benefit from innovative techniques are the following.

- 1) Development of an innovative architecture for a distributed system using the *Clouds* approach. A hierarchy of clusters seem to be the best approach. Each cluster should be a distributed system which is object based and supports atomic actions. Intracluster machine independence and fault tolerance is necessary.
- 2) In case of detected faults or emergency faults, corrective measures include recovery and reconfiguration. Recovery (or short term reconfiguration) ensures that vital functions of the systems keep ticking, no matter what happened. Also long term reconfiguration techniques would help in fault isolation, repair and recommission of faulty components.
- 3) Redundancy or replication is a time honored method of achieving reliability. In some systems, reliability is of high concern and this traditional scheme, coupled with state of the art techniques will give rise to the best reliability/performance figures. How to integrate the two is not a trivial problem, and need investigation.
- 4) The primary/backup paradigm is a powerful technique for fault tolerance. This needs extension to handle more situations than just one failure at a time. Also the probes needed to implement this scheme can be used for a variety of other purposes. There seems to be some good research directions here that need to be explored.
- 5) The *Clouds* protection mechanisms will be used to support multilevel security. The prototype does not currently support broadcast medium encryption and digital signatures. These schemes will be designed, built and tested, along with testing of their effect of general system performance. The tradeoff issues have to be identified, and the critical messages that need digital signatures have to be formally defined.

This scheme can then be extended to public broadcast channels using data encryption and digital signatures for guards against unauthorized eavesdropping, message forging and penetration of the systems.

- 6) Implementation of applications subsystems, like the database system outlined above that effectively uses *Clouds* support for enhancing system utility.

4. References

- [Allc83a] Allchin, J. E., *An Architecture for Reliable Decentralized Systems*, Ph.D. Thesis, School of Information and Computer Science, Georgia Institute of Technology, 1983 (also released as technical report GIT-ICS-83/23)
- [Allc83b] Allchin, J. E., and M. S. McKendry, *Synchronization and Recovery of Actions*, Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), Montreal, August 1983
- [Alme83] Almes, G. T., A. P. Black, E. D. Lazowska, and J. D. Noe, *The Eden System: A Technical Review*, University of Washington Department of Computer Science, Technical Report 83-10-05, October 1983
- [Brwn83] Browne J. C., et. al., *Zeus: An Object-Oriented High Integrity Distributed Operating System*. Technical Report, U of Texas Austin.
- [DaLeSp85] Dasgupta P., LeBlanc R. and Spafford E. *The Clouds Project: Designing and Implementing a Fault Tolerant, Distributed Operating System*. Georgia Tech Technical Report #GIT-ICS-85/29
- [Da86] Dasgupta P., *A Probe-Based Fault Tolerant Scheme for the Clouds Operating System* Georgia Tech Technical Report #GIT-ICS-86/05 (to appear in the Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications.)

- [DaMo86] Dasgupta P. and Morsi M., *An Object-Based Distributed Database System Supported on the Clouds Operating System* Georgia Tech Technical Report #GIT-ICS-86/07.
- [Jens82] Jensen E. D. *Decentralized Executive Control of Computers* Proc. 3rd Intl. Conf. on Distributed Computing Systems, Oct 1982, pp 31-35.
- [Jones79] Jones, A. K., *The Object Model: A Conceptual Tool for Structuring Software*, Operating Systems: An Advanced Course, Springer-Verlag, NY, 1979, pp. 7-16
- [Lisk83a] Liskov, B., and M. Herlihy, *Issues In Process and Communications Structure for Distributed Programs*, Proceedings of the Third Symposium on Reliability in Distributed Software and Database Systems, Clearwater Beach, Florida, October 1983
- [Lisk83b] Liskov, B., and R. Scheifler, *Guardians and Actions: Linguistic Support for Robust, Distributed Programs*, ACM TOPLAS, Vol. 5, No. 3, July 1983
- [McKe82] McKendry M. S. and Allchin J. E. *Object-Based Synchronization and Recovery* Technical Report GIT-ICS-82/15, Georgia Instt. of Tech.
- [McKe84b] McKendry, M. S., *Ordering Actions for Visibility*, Proceedings of the Fourth Symposium on Reliability in Distributed Software and Database Systems, Silver Spring, Maryland, October 1984
- [Schn85] Schantz R.E. and Thomas R. H. *CRONUS, A Distributed Operating System: Functional Definition and System Concept*. BBN Technical Report 5879.
- [Spec84] Spector A. Z., et. al. *Support for Distributed Transactions In the TABS Prototype*, Technical Report, CMU-CS-84-132.
- [WeLi83] Weihl, W. and B. Liskov, *Specification and Implementation of Resilient, Atomic Data Types*, Symposium on Programming Language Issues in Software Systems, June 1983
- [Wulf74] Wulf, W., et. al., *HYDRA: The Kernel of a Multiprocessor Operating System*, Communications of the ACM, Vol. 17, No. 6, June 1974.

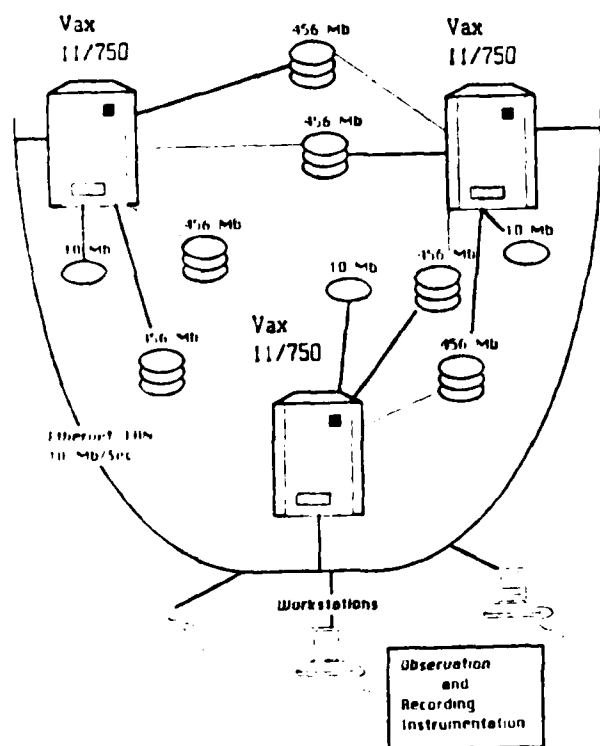


Fig. 1: The Clouds Hardware Architecture.

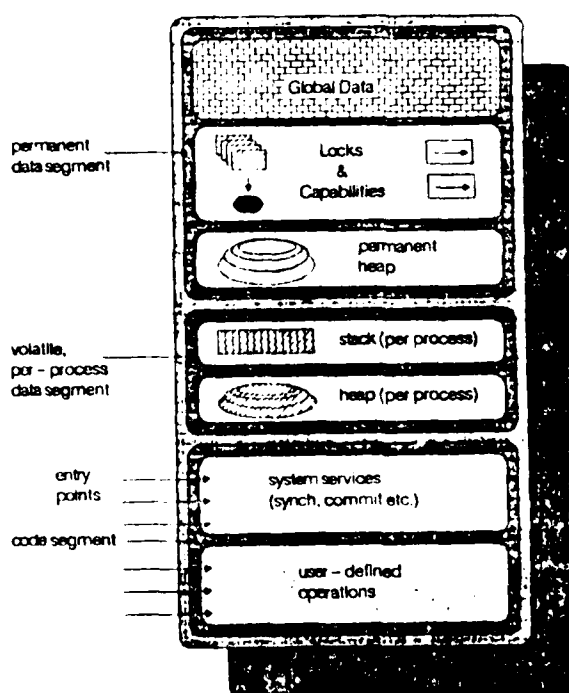


Fig. 2: Clouds Object Structure.

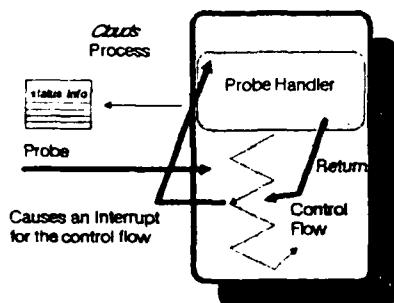
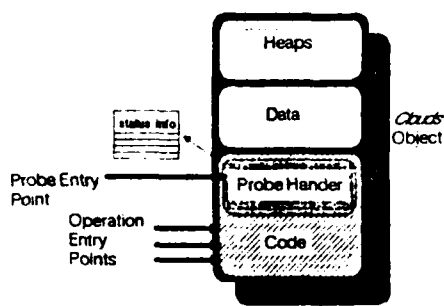


Fig. 3: Probe Handlers in Objects and Processes.

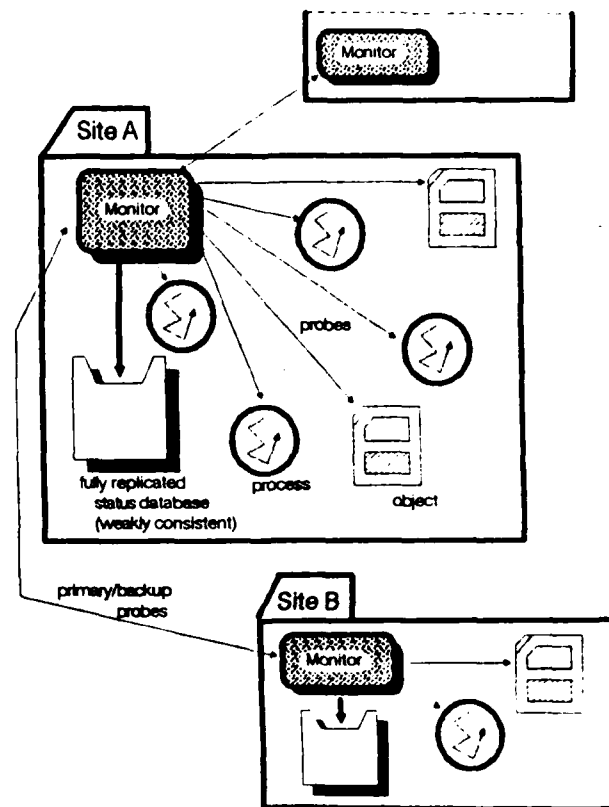


Fig4: The structure of the monitoring system.

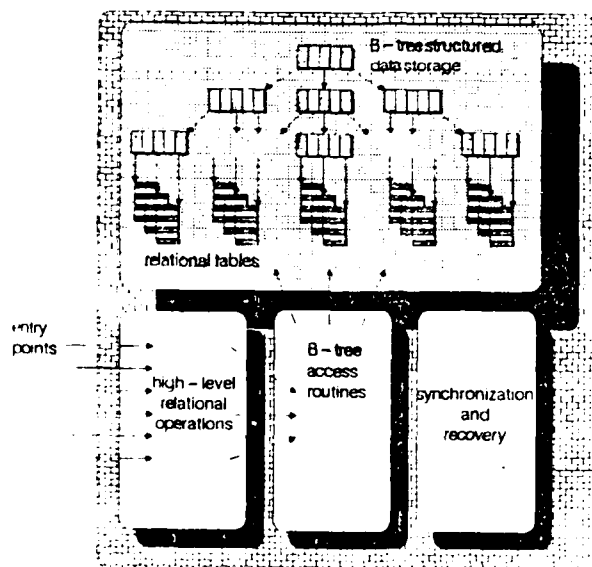


Fig. 5 A Relational Object, with B - tree data storage.

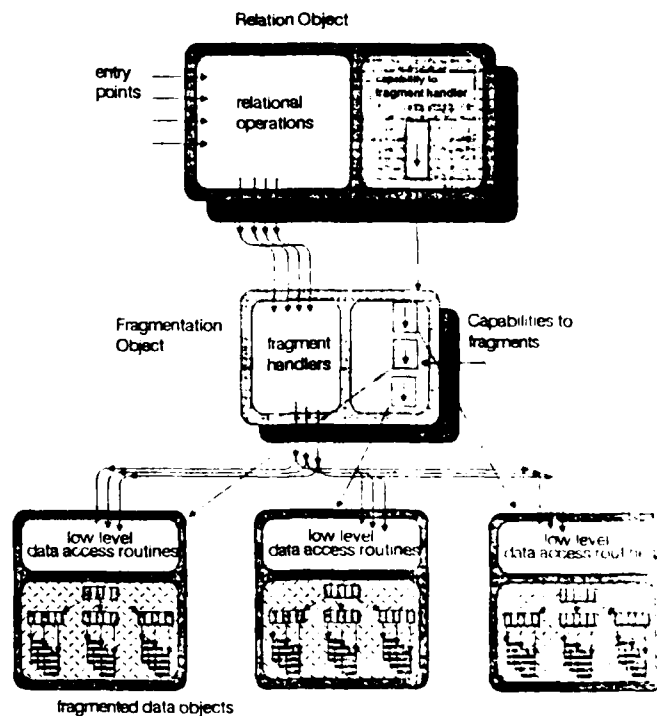


Fig. 6: The Fragmentation Handling Scheme

Session 7: Grainularity Issues

Chairperson: Jane Liu
University of Illinois at Urbana

A Large-Grain Dataflow Architecture

Ian Kaplan, Loral Instrumentation

8401 Aero Dr. San Diego, CA 92123

USENET: {ucbvaz,decvaz,ihnp4}!sdcsvaz!sdcc6!loral!ian

ARPA: sdcc6!loral!ian@UCSD

April 2, 1986

Introduction

This paper provides an overview of the Loral LDF 100 (tm) dataflow parallel processor. In this section we will describe the major design goals of the LDF 100. In section 1 we will discuss the LDF 100 dataflow programming model and in section 2 we will discuss the LDF 100 hardware architecture.

Biographers usually cover the childhood of their subject since an understanding of the person's beginnings are important in understanding what they became. Just as it is important to understand the childhood of a historical figure, the examination of a computer architecture requires an understanding of what the designers of a computer system attempted to achieve.

The LDF 100 was designed for applications in high speed real time processing (e.g., telemetry and signal processing) and numeric simulation. Listed below are some of the major design goals of the LDF 100:

Software

- The parallel processor must use a large grain dataflow programming model.
- It must be easy to program the parallel processor. This includes a good software development environment (i.e., a UNIX development environment).
- It must be possible to use standard "off the shelf" compilers for standard languages like C and Fortran.

Hardware

- The system must use "off the shelf" hardware (i.e., no custom VLSI)
- The parallel processor must provide a clean interface to the existing Loral Instrumentation telemetry system (the ADS 100).
- The parallel processor must be expandable from a few processors to more than one hundred processors.

Although these design goals are divided into software and hardware categories, these categories are not necessarily exclusive. The construction of an incrementally expandable parallel processor effects both hardware and software. The hardware must provide support for the incremental addition of processors and the software must allow a program to run efficiently on a large processors arrays.

1. The LDF 100 Software

The Dataflow Programming Model

Most parallel processors can be divided into one of two categories: those systems that use the dataflow programming model and those systems that are based on the Communicating Sequential Process (CSP) model of computation.

The dataflow model views a program as a dataflow graph. The nodes of the graph consist of the parts of the program that can be executed in parallel. These nodes are connected by arcs. Data is communicated between the graph nodes on these arcs. A node in the dataflow graph can execute as soon as it receives the necessary data.

In both a formal and a practical sense dataflow is a subset of the CSP model. A network of communicating processes can be structured like an asynchronous dataflow graph. Despite the overlap between the two models, CSP based systems are usually not structured like dataflow systems. The CSP model was originally developed to describe processes in uni-processor operating systems and most CSP based systems use a von Neumann programming model.

Compared with classically structured CSP based systems, the dataflow programming model has a number of advantages. In a dataflow program control is contained in the dataflow graph. This graph is distributed throughout the parallel processor, so control is also distributed. Dataflow graphs are asynchronous, so pipelining is easily supported. In contrast, CSP based systems are synchronous, which makes pipelining difficult.

Since the dataflow programming model is a "non-von Neumann" programming model, memory can be distributed throughout the parallel processor. The nodes of a dataflow graph process the data that they receive on their input arcs. In most dataflow systems this data arrives via a communication network or bus.

Small Grain vs. Large Grain Dataflow

Most University research has concentrated on small grain dataflow. In small grain dataflow, the nodes of the dataflow graph consist of the atomic instructions of the dataflow machine. The LDF 100 uses large grain dataflow. In this section we will discuss why the large grain model was chosen for the LDF 100.

Small Grain Dataflow

Although small grain dataflow holds out the promise of processing power that exceeds the fastest super-computers available today, no sizable small grain dataflow computer has ever been built. The problems that must be dealt with in constructing a small grain dataflow computer include:

1. The massive parallelism present in small grain dataflow demands a very high bandwidth communication media for the parallel processing elements. All of the practical small grain dataflow machines proposed so far use communication networks to support high bandwidth interprocessor communication. Networks are expensive to implement since they require custom VLSI components.
2. Small grain dataflow systems must be programmed in dataflow languages like Val, ID and Lucid. Although these are elegant languages, most prospective customers for a parallel processor prefer Fortran.
3. In many applications a number of processors will process a single large data structure. The organization of structure memory for the storage of large data structures (e.g., arrays) in dataflow systems is still a topic of research.

Large Grain Dataflow

In large grain dataflow, the dataflow graph nodes are implemented by blocks of code approximately the size of procedures. The larger granularity means that the communication bandwidth that must be supported is considerably less than that of a small grain dataflow system. Since the communication bandwidth is smaller, a segmented bus architecture can be used on the LDF 100.

A program for the LDF 100 has two components: a data graph description and a set of node implementations. The data graph is described in a custom language developed by Loral. The nodes of the graph can be implemented in *standard C* or FORTRAN. Although an existing application must be restructured, existing code can be used to implement the graph nodes.

The Loral Data Graph Language

A dataflow graph consists of a collection of interconnected nodes. The Loral Data Graph language describes the graph interconnections. It also gives a code file implementation for each node and provides some simple type information for the arcs of the graph.

```
Lymph = "hydro.code"  
import{ word[ 4 ] volume, diameter }  
export{ word[ 2 ] pressure }
```

```
Monitor = "alarm"  
import{ word[ 2 ] pressure }  
export{ word signal }
```

A graph language node is composed of four parts:

1. A node header
2. An import section
3. An export section
4. An optional firing rule

The node header contains the node name and the name of code file that implements the node. This name is a UNIX file name and is contained in double quotes.

In dataflow all nodes must have an import section or they will never be scheduled for execution. Nodes do not necessarily have to have an export sections (for example, a node that feeds an output device would have an invisible set of exports).

The import and export sections can contain multiple groups of arcs. Each group of arcs starts with a type, followed by the name of the arcs. The groups are terminated by a semicolon or the brace that ends the section.

The firing rule section is optional and is used only for dataflow nodes that implement control structures like IF-THEN-ELSE and loops.

Processors in the LDF 100 are divided between chassis. A chassis - processor combination uniquely defines a processor in the LDF 100. A program referred to as the Graph Balancer assigns a processor and a chassis to each node in the dataflow graph.

The graph from the example above, is shown here, after it has been processed by the Graph Balancer.

```
Lymph = "hydro.code" < C0> < P2>
import{ word[ 4 ] volume, diameter }
export{ word[ 2 ] pressure }

Monitor = "alarm" < C0> < P7>
import{ word[ 2 ] pressure }
export{ word signal }
```

Like most dataflow systems the LDF 100 uses a tagged token architecture. The tokens that are broadcast on the dataflow bus are 32 bits wide. The token is divided into a 16 bit data field and a 16 bit tag field. A program referred to as the Tag Assigner assigns tags to the arcs of the dataflow graph and to the code files that implement the nodes. The code files must be tagged because they are down loaded into the dataflow engine across the dataflow bus. The graph above is shown below, after it has been processed by the tag assigner.

```
Lymph = "hydro.code" < C0> < P2> < T64479>
import{ word[ 4 ] volume < T64383> , diameter < T64382> }
export{ word[ 2 ] pressure < T64381> }

Monitor = "alarm" < C0> < P7> < T64399>
import{ word[ 2 ] pressure < T64381> }
export{ word signal < T64380> }
```

Data Graph Nodes

The Loral dataflow programming environment is a highly modular one. Each node interacts with other nodes only through input and output arcs. This modular design allows code to be easily reused in new graphs. Nodes written in different languages can be intermixed without difficulty since all communication between nodes takes place via tokens broadcast on the dataflow bus.

In the dataflow environment a node will not be scheduled for execution until it has all the data it needs to execute. A node reads the data that arrived on its input arcs with a *flore*ad system call. Unlike reads in a CSP based system, *flore*ad is not a waiting read, since the data is already present when the *flore*ad is executed (otherwise the node would not have been scheduled). The C version of the *flore*ad call is shown below.

```
num_tokens = floread( ARC, &ptr );
```

The *flore*ad call returns the number of 16 bit data values read. It is passed the arc to read and a pointer to a pointer. The dataflow kernel that runs on the local dataflow processing element will initialize this pointer with the address of the data that arrived on the input arc.

Nodes output data using flowrite calls. An example of the C version of flowrite is shown below.

```
flowrite( ARC, ptr, size );
```

This flowrite call will write *size* 16 bit words from the data structure pointed to by *ptr* to the output arc *ARC*.

Firing Rules

This paper provides only a brief overview of the Loral Data Graph Language. The Data Graph Language also includes firing rules, which will not be described in this paper.

Firing rules allow the program to alter the normal conjunctive semantics of the data graphs described so far. A node that uses a conjunctive firing rule will only fire when it has *all* its inputs. To implement conditional flow of data (a dataflow control structure similar to the IF statement) and loops, more complex firing rules are needed.

2. The LDF 100 Hardware

The LDF 100 logically consists of three sub-systems:

1. The UNIX system
2. The dataflow engine
3. The real time I/O sub-system

Although these components are logically separate, they are physically packaged in one or more standard instrument rack chassis.

The UNIX System

The compilers for C and Fortran, and the dataflow graph processing tools needed to create a dataflow program are supported under UNIX. The LDF 100 uses the National Semiconductor GENIX port of UNIX for the National 32000 microprocessor. This UNIX system runs on a single board, with 1M byte of RAM, hardware floating point and demand paged virtual memory.

The Dataflow Engine

To the programmer on the UNIX system the dataflow engine appears to be a device. This device can be written to and read from, like any other device. I/O services to support reading and writing of tagged data streams are supported.

The connection between UNIX and the dataflow engine is managed by a board referred to as the FLObus (tm) Interface Processor, or FIP. The FIP is an intelligent board that has its own NS32000 processor, local RAM and a mail box ram that is used to communicate with UNIX.

The Dataflow Processor

The dataflow processor is divided into two sections:

- The Token Processing Section (TPS)
- The Node Processing Section (NPS)

The Token Processing Section (TPS) is connected to the dataflow bus. Hardware on the TPS fetches selected dataflow tokens from the dataflow bus and places them in an input FIFO. The NS32000 microprocessor in the TPS gathers the tokens from the FIFO and collects them into data sets referred to as packets. A packet consists of one or more tokens and is the amount of data needed to "complete" a node's input arc. When all of a node's input arcs are complete, the node can be scheduled for execution.

When all of the data that is needed to execute a node has been collected, the node is scheduled for execution on the Node Processing Section (NPS). Like the TPS, the NPS uses a NS32000 microprocessor. The NPS also includes a floating point unit, an interrupt control unit and a memory management unit. The NPS and TPS sections of the node processor are shown in the block diagram in figure 1.

A separate data fetch section (the NPS) and a separate application execution section (the TPS) allows pipelining within the node processor. While the NPS is executing a node the TPS can continue to collect data for the remaining nodes. Since data fetch is a bottleneck in parallel processors, this optimizes the performance of the LDF 100.

Building Large Dataflow Systems

All LDF 100 systems consist of a master chassis that contains the UNIX development environment and optional expansion chassis. These chassis are packaged in an attractive unit based on a standard instrument rack. The master chassis can hold up to eleven dataflow processors (i.e., node processors). The expansion chassis can hold up to fifteen processors each. Expansion chassis can be added to the system until it consists of more than one hundred processors.

The chassis are connected via a board referred to as the FLObus Network Interface. This board supports the fast transfer of dataflow tokens between chassis. This transfer takes place via combinatoric logic, so a delay of only three bus cycles is introduced. The FLObus Network Interface (FNI) is programmed to allow selected tokens to flow between chassis. Only those tokens that cross between chassis on interchassis graph arcs are programmed into the FNI, so the loading on the dataflow bus is minimized. A multi-chassis LDF configuration is shown in figure 2.

Acknowledgements

The LDF 100 has been developed by a small group of dedicated computer professionals. These include Jim Mees, who developed the dataflow kernel, Greg Hutchins, who did the UNIX port and who also worked on the kernel. The LDF 100 hardware was designed and developed by Lorraine M. Thompson, Gil Urcheck and Gale Williamson. John Van Zandt provided project leadership. Space does not allow us to list the many other people at Loral who have helped to make the LDF 100 a reality.

NODE PROCESSOR . . .

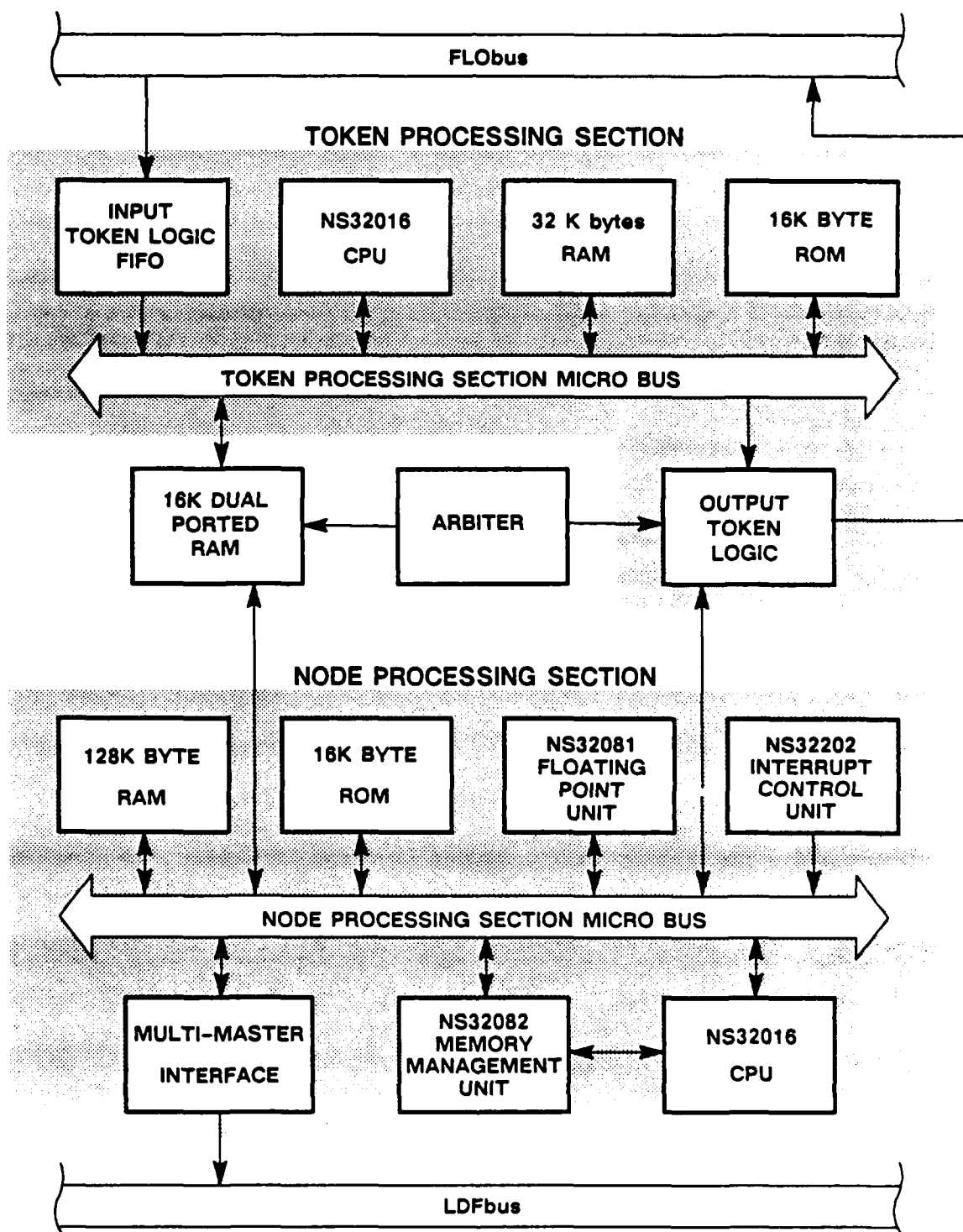


figure 1

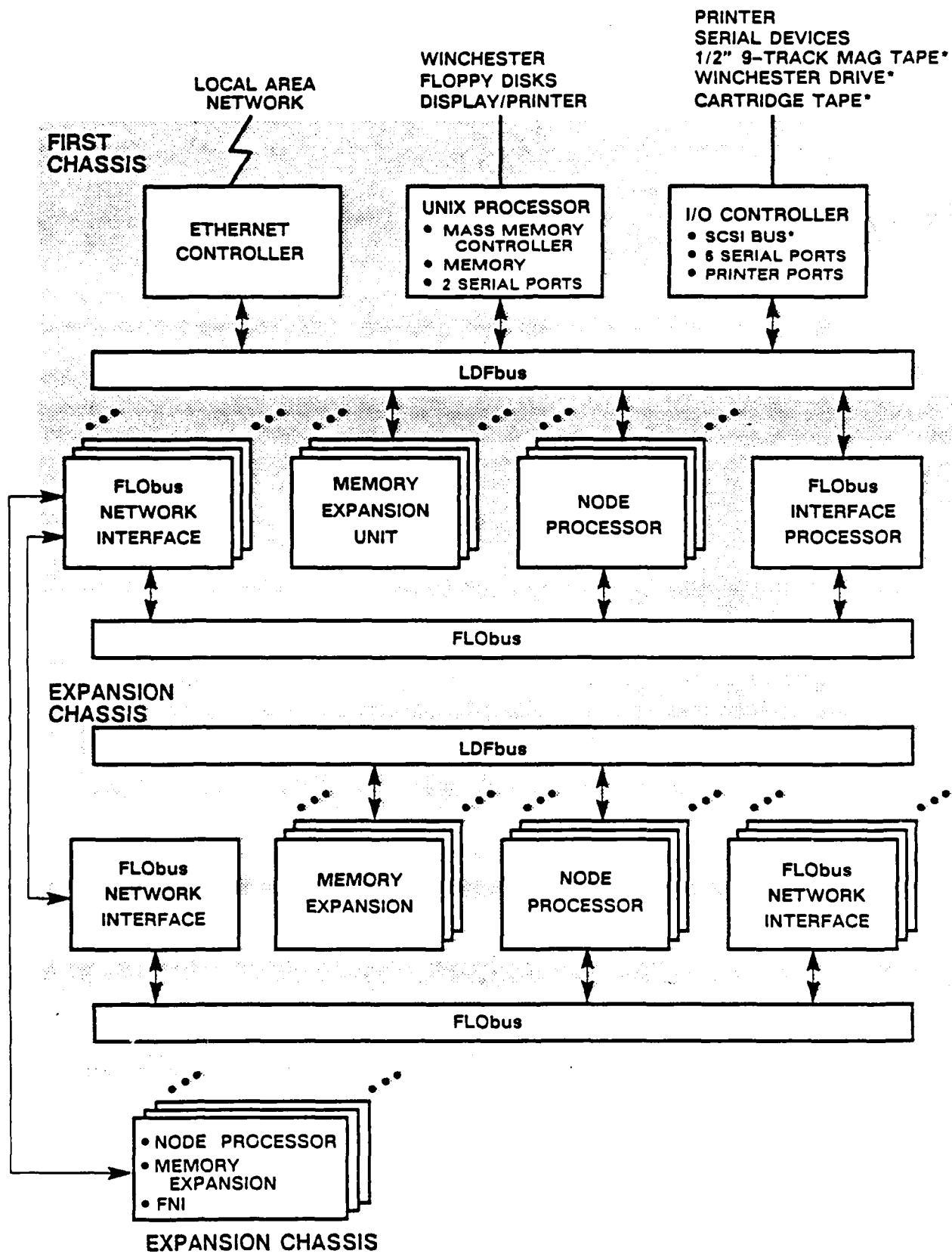


figure 2

SISAL: Initial MIMD Performance Results

R R. Oldehoeft, D. C. Cann and S. J. Allan
Computer Science Department
Colorado State University

1 Introduction

SISAL (Streams and Iteration in a Single-Assignment Language) [MSA*85] is a language for expressing algorithms to execute on highly concurrent computers. Like other functional languages, SISAL displays characteristics allowing compilers to detect and easily exploit the underlying architectural parallelism and the potential for parallelism in the program.

Four groups associated with different organizations and with different original target architectures cooperated to define SISAL. The organizations (and architectures) are: Lawrence Livermore National Laboratory (Cray vector processors), Digital Equipment Corporation (VAX¹ processor clusters), the University of Manchester (Manchester data flow machine), and Colorado State University (Denelcor HEP multiprocessor). SISAL descends from VAL [AD79,McG82] and retains its functional, single assignment character. SISAL differs from VAL in possessing simpler error types, general recursion, a stream data type, and improved iteration forms. The SISAL project has also benefited from earlier work described in [GP79] and [KLP79].

SISAL offers several advantages for use on a multiprocessor system. The user need not (and cannot) manage or express parallelism in a program. The compiler detects parallelism, decides how much to exploit, and generates code to take advantage of the parallelism and manage it. Users express algorithms without expressing or implementing parallelism. Language syntax is Pascal-like. A familiar syntax decreases the time it takes to learn the language and aids in program readability. For a SISAL language overview see [AO85].

The SISAL groups have formed a pool of benchmark programs typical of those that programmers would actually write using SISAL. Many are short, but there are some over 1,500 lines long. Thus they often represent actual codes run in the "real world."

At CSU we implemented SISAL on the Denelcor HEP and evaluated the performance of the initial implementation. Work proceeds to port the software to other MIMD systems. The sections below briefly discuss the compiler implementation with emphasis on portability considerations, describe the run time support system, and survey performance results. See [AO86] for a more complete description of the SISAL implementation at CSU.

¹VAX is a trademark of Digital Equipment Corporation

	Input		Translator		Output
(1)	SISAL program	→	SISAL parser	→	IF1
(2)	IF1	→	IF1 optimizer	→	IF1 (optional)
(3)	IF1	→	IFPCC generator	→	IFPCC
(4)	IFPCC	→	C code generator	→	Native code

Figure 1: SISAL Programming System at CSU

2 Compiler

The SISAL compiler, designed for flexibility and portability, consists of several phases. We use a parser that produces a machine-independent intermediate form, "IF1" [SG85], along with optimizers that improve this intermediate form [SG85]; both were developed at LLNL. The next phase [CAO84a] translates to another intermediate form acceptable to the second pass (code generation and optimization) of the portable C compiler [Joh78], used almost universally in UNIX² systems. This approach has saved a great deal of effort and promises to ease the porting of the translator system to other parallel processors that support C, or facilitate its use as a cross-compiler. A diagram of the CSU SISAL compiler appears in Figure 1.

3 Run Time System

Most SISAL run time software is written in C, with a few assembly language routines. This aids in system portability to other machines running UNIX. The major responsibilities of the run time routines are managing processes, arrays, streams, and dynamic storage.

Parallel execution units in MIMD SISAL are function bodies, parallel loop "stripes," and multi-expression components. Execution of parallel loops and multi-expression components is performed by daemon processes that interact closely with SISAL process management in low-overhead ways. (For these tests, this had to be emulated by packaging loop slices as explicit functions; the automatic capability is now fully implemented.) So mapping SISAL processes onto the hardware resources is the major concern. If each function reference occupied a processor, then either the number of available processors would be overrun or deadlock would result as parent processes held processors waiting for completion of child functions who cannot execute. We prevent deadlock by requiring that a parent process waiting for value(s) from children relinquish its processor and suspend execution. See [BAO84] and [VAO85] for more complete discussions of process management.

Multi-dimensional arrays in SISAL are "arrays of arrays" and are dynamic. At compile time we know only the dimensionality and element type; bounds are in general determined only at run time. Necessary resulting run time tasks include adding an element to an array, fetching a value from an array, etc. [CAO84b] has a complete discussion on array implementation.

²UNIX is a trademark of AT&T Bell Laboratories.

Streams are an important data structure because they provide pipelined parallelism among stream producer and consumer functions. Both the stream producer and consumers execute simultaneously so the program processes can form generalized parallel pipelines. Only a contiguous substream of the stream needs to be extant at any time because values that the slowest stream consumer no longer will reference need not remain. In this way programs can execute that produce and consume (finite prefixes of) infinite streams; these programs terminate normally. For a further discussion see [AO83].

SISAL run time support relies on dynamic storage allocation for process descriptors, streams, and arrays. An adapted boundary tag method is in place that allows multiple concurrent allocations and deallocations [BAO85]. Also, we include a "front end" caching scheme that can greatly speed up dynamic storage management under the conditions that obtain during SISAL execution [OA85a]. For a complete discussion of dynamic storage management in SISAL, as well as other run time support details, see [OA85b].

Note that in these tests array storage is not properly recycled, adversely affecting performance. We anticipate that a new intermediate form under development at LLNL will help all SISAL implementations to optimize storage management.

4 Performance

We briefly describe the performance of a few benchmark programs as measured on a single-PEM Denelcor HEP processor; unfortunately this machine is no longer available. Because of the implementation restrictions mentioned in the previous section, we believe these results represent a baseline for enhanced performance on other shared-memory systems. Overall, the speedup curves display performance similar to those of HEP programs written in imperative languages with explicit, low-level process synchronization mechanisms. Further, these curves follow the general form predicted for this architecture in [Jor85].

The performance of a simple numeric integration program is seen in Figure 2. The speedup curve is typical of programs that operate on scalars. Process management among SISAL functions that exchange simple values is being tested here.

Figure 3 shows the speed up curve for a matrix multiplication program. The sawtooth effect seen in the speedup curve comes from varying the number of hardware resources against a fixed size for the data. The periodic degradations are due to the lack of array space recycling.

A noise filtering program was coded in SISAL; results are displayed in Figure 4. The sawtooth effect of the previous benchmark is not as pronounced because storage interference is not a factor.

Finally, the performance of a sieve method for prime number generation is given in Figure 5. This graph shows the best speedup among the four for small numbers of hardware processes. Beyond 13 hardware processes there is not enough work to keep SISAL functions running and offset the cost of frequently idling hardware processes; performance actually degrades. The smoothness of the curve shows that stream run time support operates in a predictable fashion.

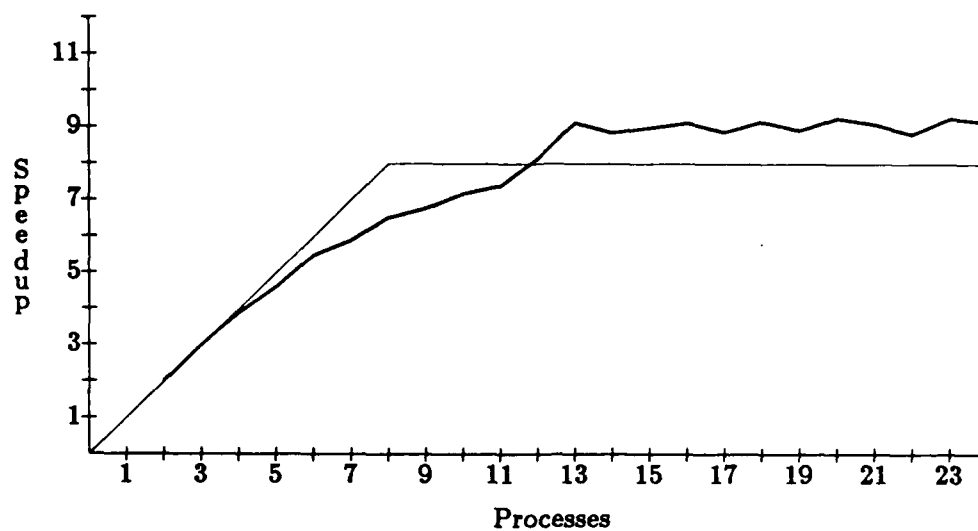


Figure 2: Numerical Integration Performance Results

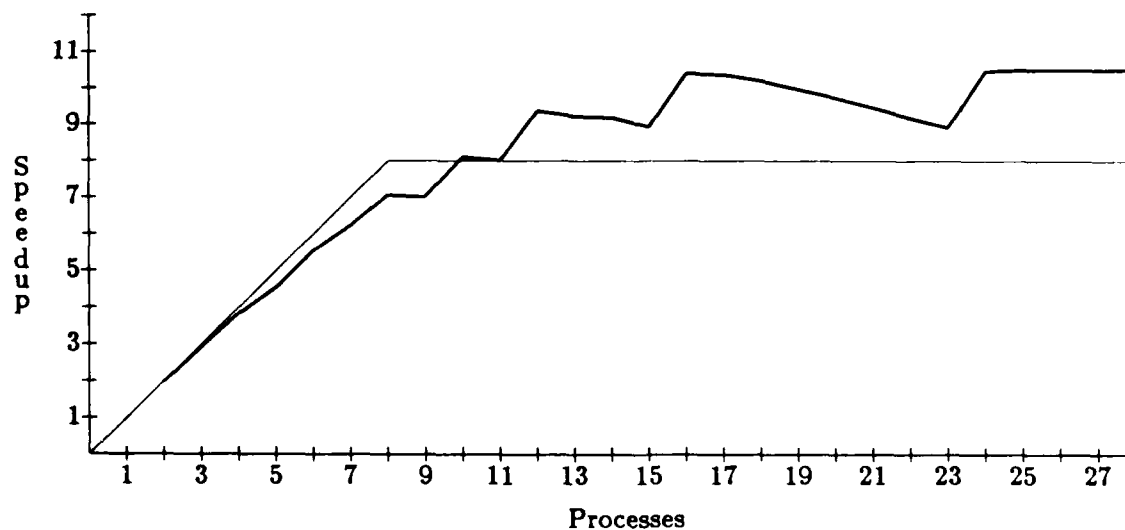


Figure 3: Matrix Multiplication Performance Results

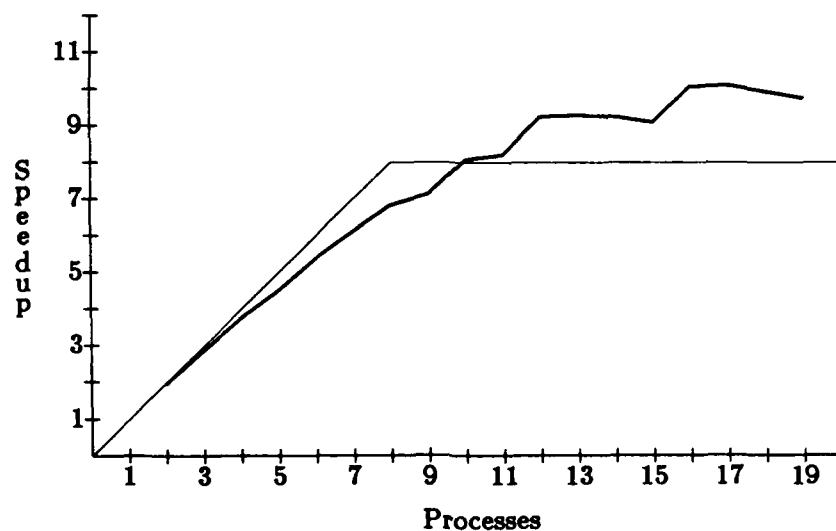


Figure 4: Noise Removal Performance Results

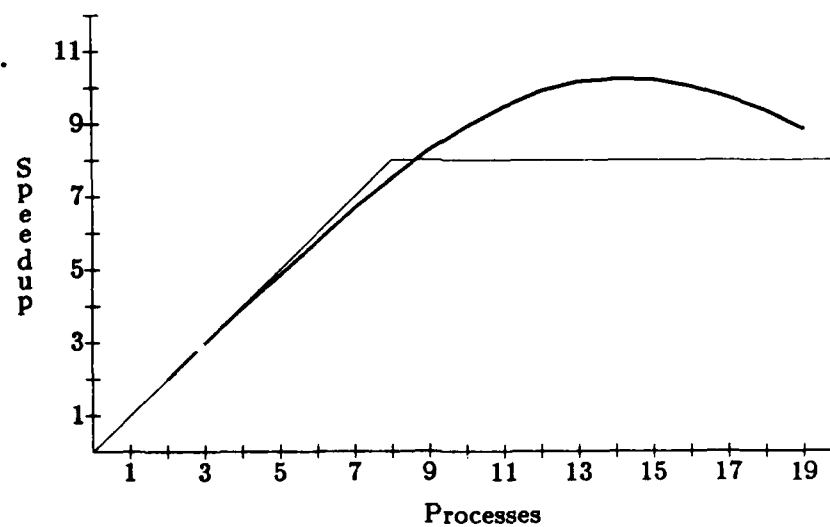


Figure 5: Sieve of Eratosthneses Performance Results

References

- [AD79] William B. Ackerman and Jack B. Dennis. *VAL - A Value-Oriented Algorithmic Language*. Technical Report LCS/TR-218, MIT, June 1979.
- [AO83] Stephen J. Allan and R. R. Oldehoeft. A stream definition for von Neumann multi-processors. In *Proceedings of the 1983 International Conference on Parallel Processing*, pages 303-306, August 1983.
- [AO85] Stephen J. Allan and R. R. Oldehoeft. HEP SISAL: parallel functional programming. In J. Kowalik, editor, *Parallel MIMD Computation: The HEP Supercomputer and Its Applications*, pages 123-150, MIT Press, Cambridge, MA, 1985.
- [AO86] Stephen J. Allan and R. R. Oldehoeft. Parallelism in SISAL: Exploiting the HEP architecture. In *19th Hawaii International Conference on System Sciences*, pages 538-548, 1986.
- [BAO84] Larry W. Booker, Stephen J. Allan, and R. R. Oldehoeft. *Process management for HEP SISAL*. Technical Report CS-84-05, Colorado State University Computer Science Department, Fort Collins, CO, June 1984.
- [BAO85] Bruce Bigler, Stephen J. Allan, and R. R. Oldehoeft. Parallel dynamic storage allocation. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 276-279, August 1985.
- [CAO84a] David C. Cann, Stephen J. Allan, and R. R. Oldehoeft. *An IF1 Driven Portable Code Generator*. Technical Report CS-84-15, Colorado State University Computer Science Department, Fort Collins, CO, December 1984.
- [CAO84b] Steven Cobb, Stephen J. Allan, and R. R. Oldehoeft. *Arrays in SISAL*. Technical Report CS-84-04, Colorado State University Computer Science Department, Fort Collins, CO, June 1984.
- [GP79] D. Grit and R. Page. *A multiprocessor model for parallel evaluation of applicative programs*. Technical Report, Colorado State University, Fort Collins, CO, September 1979.
- [Joh78] S.C. Johnson. A portable compiler: theory and practice. In *Conference Record of the 5th ACM Symposium on the Principles of Programming Languages*, pages 97-104, ACM, New York, January 1978.
- [Jor85] Harry F. Jordan. HEP architecture, programming and performance. In J. Kowalik, editor, *Parallel MIMD Computation: The HEP Supercomputer and Its Applications*, pages 1-40, MIT Press, Cambridge, MA, 1985.
- [KLP79] R. Keller, G. Lindstrom, and S. Patil. A loosely-coupled applicative multi-processing system. In *Proceedings of the 1979 AFIPS National Computer Conference*, pages 613-622, 1979.
- [McG82] James R. McGraw. The VAL language: description and analysis. *ACM Transactions on Programming Languages and Systems*, 4(1):44-82, 1982.

- [MSA*85] James McGraw, Stephen Skedzielewski, Stephen Allan, Rod Oldehoeft, John Glauert, Chris Kirkham, Bill Noyce, and Robert Thomas. *SISAL: Streams and Iteration in a Single Assignment Language Language Reference Manual, Version 1.2*. Lawrence Livermore National Laboratory, Livermore, CA, M-146, rev. 1 edition, March 1985.
- [OA85a] R. R. Oldehoeft and S. J. Allan. Adaptive exact-fit storage management. *Communications of the ACM*, 28(5):506-511, 1985.
- [OA85b] R. R. Oldehoeft and Stephen J. Allan. Execution support for HEP SISAL. In J. Kowalik, editor, *Parallel MIMD Computation: The HEP Supercomputer and Its Applications*, pages 151-180, MIT Press, Cambridge, MA, 1985.
- [SG85] Stephen Skedzielewski and John Glauert. *IF1 - An intermediate form for applicative languages*. Lawrence Livermore National Laboratory, Livermore, CA, M-170 edition, July 1985.
- [VAO85] Bruce Votipka, Stephen J. Allan, and R. R. Oldehoeft. *HEP SISAL Process Management*. Technical Report CS-85-08, Colorado State University Computer Science Department, Fort Collins, CO, May 1985.

**Session 8: Mapping Algorithm and Task
Assignment**

**Chairperson: Doyce Satterfield
U.S. Army Strategic Defense Command**

THE MAPPING OF PARALLEL ALGORITHMS TO RECONFIGURABLE PARALLEL ARCHITECTURES

*Leah H. Jamieson **
*Howard Jay Siegel **
*Edward J. Delp **
*Andrew Whinston ***

** School of Electrical Engineering*
*** Krannert Graduate School of Management*
*** Department of Computer Sciences*
Purdue University
West Lafayette, Indiana 47907

Abstract

One of the significant problems which must be addressed if we are to realize the computing potential offered by parallel architectures has to do with developing a better understanding of the relationship between parallel algorithms and parallel architectures. In this paper, research on the mapping of algorithms to reconfigurable parallel architectures is presented. The thrust of this work is in identifying those characteristics of parallel algorithms which have the greatest effect on their execution, and in identifying a correspondence between those characteristics and the characteristics of parallel architectures. The context of this work is in the design of an Intelligent Operating System for the PASM reconfigurable multimicroprocessor system. The task of the Intelligent Operating System will be to direct the selection and scheduling of algorithms and the configuring of the architecture for the execution of an image understanding system.

1. Introduction

In both parallel architectures and parallel algorithms, there exist many design choices for which there are no direct counterparts in conventional serial processing. In architectures, examples of such choices include number of processors, local versus global memory organization, mode of processing (synchronous or asynchronous), and interconnection topology. In parallel algorithms, issues which do not arise in serial programming include determination of the number of processors needed/useful for a task, data allocation across memories, synchronization necessary/beneficial, and interprocessor communication requirements. The move to parallelism has introduced new degrees of freedom to both the architecture and algorithm design process. For effective use of parallel systems, it is essential to obtain a good match between algorithm requirements and architecture capabilities.

The question of mapping parallel algorithms to parallel architectures has importance at three levels. First, it bears directly on the algorithm design process. General knowledge about what constitutes an effective match between a parallel algorithm and a parallel architecture can accelerate the process of developing new parallel algorithms for a given machine. Second, an understanding of the relation between algorithms and architectures is a prerequisite for the

This research was supported by the Rome Air Development Center under Contract F30602-83-K-0119, by the Naval Research Laboratory under Grant N00014-85-C-2182, by the Air Force Office of Scientific Research under Grant F49620-86-K-0006, and by the Institute for Defense Analyses Supercomputing Research Center under Contract MDA904-85-5027.

fast, efficient design of algorithmically-specialized systems [SnJ85]. Given a fixed set of algorithms, architectures tailored for the execution of those algorithms can be developed if the architectural requirements of the algorithms are understood. Third, a general method of relating algorithms and architectures will allow efficient use of flexible parallel systems such as PASM [SiS84]. Many problems in image understanding require execution of a large number of algorithms, with the exact choice and execution sequence of the algorithms varying depending on the input data. Critical timing requirements may accompany such scenarios. Reconfigurable parallel systems can provide an environment which can be used to test the ability of various architectures -- either sequences of simulated special-purpose parallel systems or successive configurations of the reconfigurable system itself -- to meet these computing needs. Integral to the effective use of these flexible parallel systems, however, will be the ability to select machine configurations based on knowledge about the algorithms to be executed. In order to accomplish this automatically, the operating system will need to use information about the characteristics of the algorithms to select successive configurations of the parallel architecture.

In [DeS85], we have presented a model for an Intelligent Operating System for the execution of image understanding tasks on PASM (see Fig. 1). In this model, the Algorithm Database contains the algorithms which can be used in performing the image understanding task, plus two types of information: properties of the algorithm which describe its image analysis capabilities (e.g., "edge finding in noisy images") and properties which describe the execution characteristics of the algorithm on different architecture configurations. The Algorithm Database may contain multiple algorithms to perform the same basic task (e.g., edge finding), with different versions providing different image processing capabilities (e.g., good performance in the presence of noise) and/or having different parallel implementations (e.g., based on raster versus square subimage allocation of the image to the parallel memories). The Intelligent Image Understanding System determines what types of symbolic and iconic operations should be performed, selects the algorithms to be executed, and uses the results from these operations to determine what needs to be done next. The Reconfigurable Parallel Processing System in our study is PASM, which can be dynamically reconfigured under software control to operate as one or more independent partitions ("virtual machines") of various sizes. Each partition can operate in either SIMD or MIMD mode, and can dynamically switch modes under software control. The Low-Level Operating System Routines accomplish the actual reconfiguration. The Intelligent Operating System uses information from the Intelligent Image Understanding System, Algorithm Database, and knowledge about the current system state to select a specific algorithm implementation and machine configuration for accomplishing the next step of the image understanding task. The Intelligent Image Understanding System and the Intelligent Operating System are to be implemented as expert systems.

Various researchers have examined the problem of characterizing parallel architectures (e.g., see [Kun80, EtN83]), characterizing parallel algorithms (e.g., see [DaL81, HoJ81, SmS85]), and relating algorithms to architectures (e.g., see [CaL82, ChF83]). In this paper, we consider the problem of identifying the characteristics of parallel algorithms which have the greatest effect on their execution, and of identifying a correspondence between those characteristics and the characteristics of parallel architectures. This information is to be used in constructing the Algorithm Database in the above model.

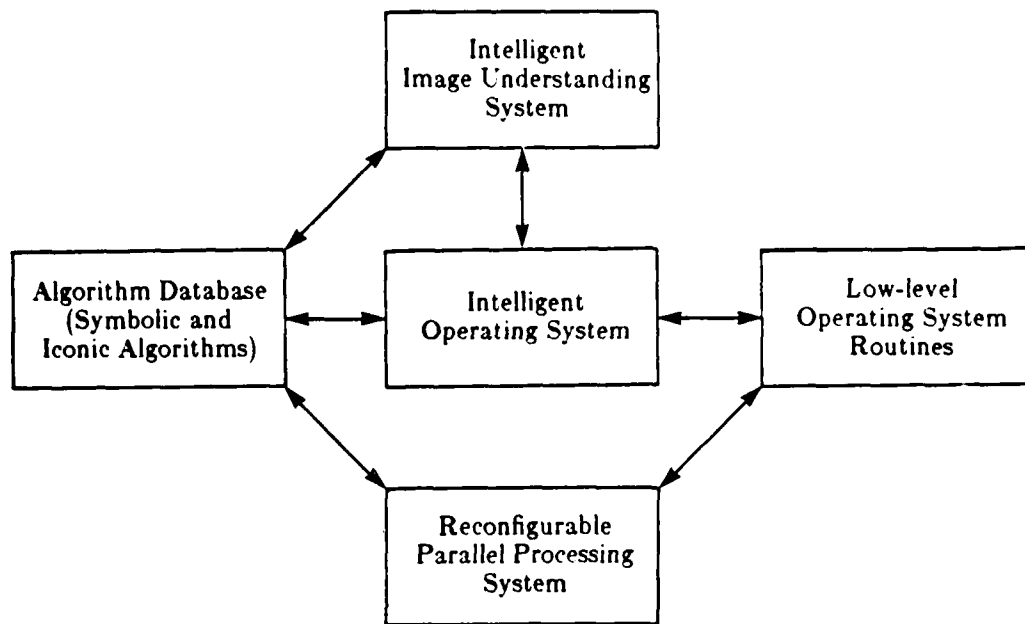


Fig. 1. System model.

2. Architecture Model

We assume a target parallel architecture with the general attributes listed below. The purpose of the assumptions is to define a very general architecture framework, so that no undesirable restrictions are imposed on the architecture by the assumptions. Rather, by examination of the algorithm attributes, the subset of capabilities required in the architecture can be selected. Thus the framework is more general than most existing parallel systems, but many architectures, including PASM [Si84], Ultracomputer [Got82], RP3 [Pf85], the Cosmic Cube [Sei85], the Butterfly [Cro85], the Connection Machine [Hil85], MPP [Bat80], Clip 4 [Duf82], and pyramid architectures [TaK80, Uhr83], can be characterized in terms of subsets of these design attributes.

Assumptions for the target architecture are as follows:

- The system consists of a large number of homogeneous processors. In mapping the algorithm to an ideal parallel architecture, the number of processors available is considered to be unlimited.
- The system can be organized with processors accessing a shared global memory or with each processor having an associated local memory, or a hybrid of the two approaches.
- The system is partitionable into independent submachines of various sizes. This implies that (1) a number of algorithms can be executing simultaneously, on different partitions of the system, and (2) for each algorithm, the partition size can be selected to meet the needs of the algorithm. The partitioning can be changed dynamically at execution time.
- Each partition of the system (and the entire system itself) is capable of both SIMD and MIMD operation, and can dynamically switch between modes during execution.
- The system has a flexible interconnection network which can provide a wide variety of communications patterns within each partition.

Given the above assumptions, the problem becomes one of selecting the architecture configuration – memory organization, partition size, mode of operation, network configuration

-- which best matches the attributes of the algorithm.

3. Algorithm Characteristics

Candidate algorithm characteristics are listed below. Table 1 summarizes the relationship between algorithm and architecture characteristics. A "1" entry in the table indicates a probable primary dependence between the algorithm and architecture characteristics, an entry of "2" indicates a likely secondary dependence and "3" denotes a less strong dependence.

1. *Nature of the parallelism.* Under this category come a number of attributes having to do with the "kind" of parallelism which is used and the way in which the algorithm and/or data can be decomposed.
 - a. *Data parallelism versus functional parallelism.* Parallelism can be achieved by dividing the data among the processors, by decomposing the algorithm into segments which can be assigned to different processors, or by macropipelining (which is a special case of decomposition of the algorithm into functional segments). The type of parallelism will affect the allocation of data, the assignment of processes to processors, and the basic decision as to what mode of parallelism (SIMD/MIMD/macropipeline) to use.
 - b. *Data granularity.* Data granularity deals with the "size" of the data items processed as a fundamental unit, and will have a bearing on the data allocation, communications requirements, processor capability, and memory requirements.
 - c. *Module granularity.* Module granularity [Kun80] quantifies the amount of processing which can be done independently, either of other processes or of operations being performed in other processors. It is essentially a measure of the frequency of synchronization, and will affect the choice of SIMD versus MIMD operation, the assignment of processes to processors, the communications requirements, and the likelihood of equalizing the execution times of component parts of the algorithm.
2. *Degree of parallelism.* This will be related to both the data granularity and the module granularity. Its most direct impact will be on the choice of machine size and on the maximum speedup attainable.
3. *Uniformity of the operations.* If the operations to be performed are uniform (e.g., across the data or feature set), then SIMD (or pipeline) processing may be feasible. If the operations are not uniform, then MIMD processing will be chosen and strategies to equalize the computational load across the processors may come into play. These strategies may be applied statically at compile time or dynamically at execution time.
4. *Synchronization requirements.* In addition to the synchronization requirements implied by the process granularity, consideration of precedence constraints is implicit in characterizing the synchronization requirements. This will affect the assignment of processes to processors and the scheduling of various components of the algorithm.
5. *Static/dynamic character of the algorithm.* The pattern of process generation and termination will affect the processor utilization, the scheduling of sub-processes, the memory organization, and the communications requirements.
6. *Data dependencies.* The data dependencies in an algorithm will play the largest role in dictating data allocation patterns and communications characteristics. They will also have a major part in the decision to use a global versus local memory organization.
7. *Fundamental operations.* The basic operations performed in the algorithm will dictate the processor capabilities needed.

Table 1. Relationship Between Algorithm and Architecture Characteristics

Algorithm Characteristics	Architecture Characteristics										
	Number of PEs	Memory Organization	Memory Size	Mode (SIMD/MIMD/Pipe)	Network	Synchronization	Processor Capability	Data Types	Addressing Modes	Data Structures	I/O
	Type of Parallelism	3	2	3	1		3				
	Data Granularity		1	2	1	3				2	
	Module Granularity				1	1	2				
	Degree of Parallelism	1			2						
	Memory	2	2	1							
	Uniformity		2		1	3					
	Synchronization		2		1	2	1				
	Static/Dynamic		1		1	2	3				
	Data Dependencies		2		3	1					
	Fundamental Ops.						1				
	Data Types					3		1	2		1
	Data Structures		2		3	2			3	1	
	I/O		3	3	2	2					1

8. *Data types and precision.* The atomic data types and data precision will bear most directly on the individual processor capability and on the memory requirements, but may also imply requirements for communications bandwidth.
9. *Data structures.* Many algorithms can be characterized as having a "natural" data structure (or structures) on which operations are performed. The ability of an architecture to support the needed access patterns, to exploit possible regularity in the structures, and to allow the needed interactions between parts of the structures will affect algorithm performance.

4. A Model for Verifying the Algorithm Characteristics Set

Using the model from [DeS85] as a basis (Fig. 1), we present a model for verifying the usefulness of the algorithm characteristics set for mapping algorithms to architectures. The system will include five components, as shown in Fig. 2: an Algorithm Characteristics Database, an Architecture Configurations Database, a Reconfigurable Parallel Processing System (i.e., the actual architecture), a set of Low-level Operating System Routines, and the Configuration Selector itself.

- The Algorithm Characteristics Database will contain a compendium of information about the characteristics of parallel algorithms and their relations to the various aspects of parallel architectures.
- The Architecture Configurations Database will contain the parameters which describe the underlying Reconfigurable Parallel Processing System. This will include information about the number of processors in the system, possible memory organizations which can be selected, modes of operation which are available, and a description of the interconnection network. In the most general case, this will correspond to the architecture assumptions outlined in Section 2. When running the operating system on a particular architecture, it will contain detailed information about that architecture.
- Our Reconfigurable Parallel Processing System will be PASM. PASM satisfies the architecture assumptions outlined in Section 2 above, and is therefore a suitable target architecture. The availability of the PASM simulator and prototype will allow us to test our architecture-dependent algorithms on the machine configurations selected by the operating system.
- The Low-level Operating System Routines will be used to do the actual system configuration, and are currently under development for PASM.
- The Configuration Selector will select the machine configuration given the characteristics of the parallel algorithm and the parameters of the Reconfigurable Parallel Processing System. The function of the Configuration Selector will be to relate the characteristics of a parallel algorithm to possible architecture configurations, and to combine the demands of the various characteristics into a single decision about the machine configuration. Furthermore, the Configuration Selector will have to determine the assignment of resources to multiple algorithms which can be executed simultaneously. The Configuration Selector will employ the information in the Algorithm Characteristics Database and the Architecture Configurations Database. Possible tools for implementation of this component include expert systems and rule sets. (An obvious question arises as to how the characteristics of a new algorithm are obtained. A long term facet of this research will involve studying to what extent the characteristics can be extracted automatically from a high-level-language representation of the algorithm (e.g., by a compiler). For the purposes of demonstrating the algorithm-to-architecture mapping, the more immediate goal is the development of a user interface

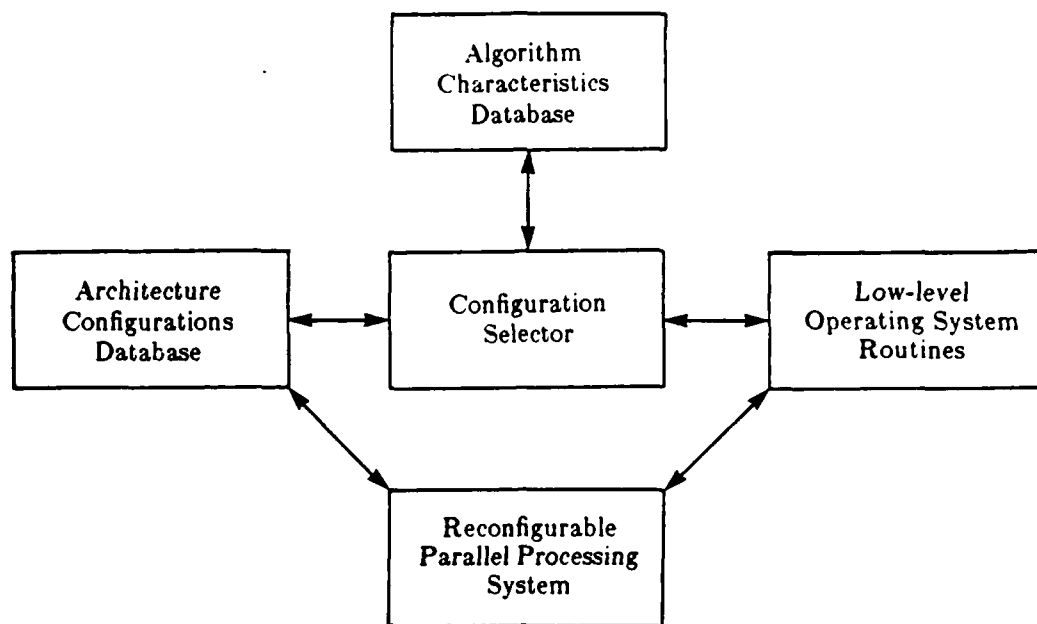


Fig. 2. Organization of the Operating System Component for Mapping Algorithms to Architectures.

which can help the user frame the algorithm in terms of the characteristics needed to allow the operating system to select an architecture configuration.)

The combination of the Algorithm Characteristics Database, the Architecture Configurations Database, and the Configuration Selector will perform the automatic selection of a machine configuration suitable for execution of the given algorithm. The Reconfigurable Parallel Processing System, the Low-level Operating System, and the Architecture Configurations Database are *machine-dependent* components of the system. The Algorithm Characteristics Database and the Configuration Selector components, however, are *machine-independent*. Thus, although we are using a specific system as a vehicle for demonstrating our work, the knowledge about the mapping of parallel algorithms to parallel architectures will be general in nature, and will be applicable to a broad class of systems.

References

- [Bat80] K. Batcher, "Design of a Massively Parallel Processor," *IEEE Trans. Comp.*, Vol. C-29, Sept. 1980, pp. 836-840.
- [CaL82] V. Cantoni and S. Levialdi, "Matching the Task to an Image Processing Architecture," *6th Int. Conf. Pattern Recognition*, Oct. 1982, pp. 254-257.
- [ChF83] Y. P. Chiang and K. S. Fu, "Matching Parallel Algorithm and Architecture," *1983 Int. Conf. Parallel Processing*, Aug. 1983, pp. 374-380.
- [Cro85] W. Crowther et al., "Performance Measurements on a 128-Node Butterfly Parallel Processor," *1985 Int. Conf. Parallel Processing*, Aug. 1985, pp. 531-540.
- [DaL81] P. E. Danielsson and S. Levialdi, "Computer Architectures for Pictorial Information Systems," *Computer*, Nov. 1981, pp. 53-67.
- [DeS85] E. J. Delp, H. J. Siegel, A. Whinston, and L. H. Jamieson, "An Intelligent Operating System for Executing Image Understanding Tasks on a Reconfigurable Parallel Architecture," *1985 IEEE Computer Society Workshop on Computer Architectures for*

Pattern Analysis and Image Database Management, Nov. 1985, pp. 217-224.

- [Duf82] M. J. B. Duff, "Parallel Algorithms and their Influence on the Specification of Application Problems," in *Multicomputers and Image Processing*, K. Preston and L. Uhr, eds., Academic Press, New York, 1982, pp. 261-274.
- [EtN83] R. D. Etchells and G. R. Nudd, "Software Metrics for Performance Analysis of Parallel Hardware," *DARPA Image Understanding Workshop*, June 1983, pp. 137-147.
- [Got83] A. Gottlieb, et al., "The NYU Ultracomputer -- Designing an MIMD Shared Memory Parallel Computer," *IEEE Trans. Comp.*, Vol. C-32, Feb. 1983, pp. 175-189.
- [Hil85] W. D. Hillis, *The Connection Machine*, MIT Press, Cambridge, MA, 1985.
- [HoJ81] R. W. Hockney and C. R. Jesshope, *Parallel Computers: Architecture, Programming and Algorithms*, Adam Hilger Ltd., Bristol, 1981.
- [Kun80] H. T. Kung, "The Structure of Parallel Algorithms," in *Advances in Computers*, Vol. 19, Academic Press, New York, 1980, pp. 65-112.
- [Pfi85] G. F. Pfister, et al., "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *1985 Int. Conf. Parallel Processing*, Aug. 1985, pp. 764-771.
- [Sei85] C. Seitz, "The Cosmic Cube," *CACM*, Vol. 28, Jan. 1985, pp. 22-33.
- [SiS84] H. J. Siegel, T. Schwederski, N. J. Davis IV, and J. T. Kuehn, "PASM: A Reconfigurable Parallel Processing System for Image Analysis," *Workshop on Algorithm-guided Parallel Architectures for Automatic Target Recognition*, July 1984, pp. 263-291. (Reprinted in *ACM SIGARCH Computer Architecture News*, Vol. 12, Sept. 1984, pp. 7-19.)
- [SmS85] B. W. Smith and H. J. Siegel, "Models for Use in the Design of Macropipelined Parallel Processors," *12th An. Symp. Computer Architecture*, June 1985, pp. 116-123.
- [SnJ85] L. Snyder, L. H. Jamieson, D. B. Gannon, and H. J. Siegel, *Algorithmically Specialized Parallel Computers*, Academic Press, Orlando, FL, 1985.
- [TaK80] S. L. Tanimoto and A. Klinger, eds., *Structured Computer Vision: Machine Perception through Hierarchical Computation Structures*, Academic Press, New York, 1980.
- [Uhr83] L. Uhr, "Pyramid Multi-computer Structures, and Augmented Pyramids," in *Computing Structures for Image Processing*, M. J. B. Duff, ed., Academic Press, London, 1983, pp. 95-112.

A Distributed System Architecture Based on Macro Dataflow Model

by

Jane W. S. Liu and Andrew Grimshaw

I. Introduction

Increasingly in recent years, distributed systems containing interconnected workstations and host computers are used to provide a wide range of services in word processing and text formatting, computing, and information management and distribution. A user on such a system typically executes his jobs on his own workstations or local host. As a result, many workstations are frequently idle and host computers are under utilized. Redundant resources naturally available in such a system are often not effectively used to improve the reliability and availability of services provided by the system. (Since it is not necessary to distinguish between them, we refer to both workstations and host computers as *hosts*.)

There are two approaches to capture and use the capability of idle or under-utilized hosts in a distributed system. In one approach, heavily loaded hosts send new jobs and migrate executing jobs to lightly loaded hosts for execution, i.e., hosts do load balancing. Many load balancing schemes have been proposed to improve response time and resource utilization [1-9]. LOCUS, Accent, V-System, and DEMOS/MP are examples of recent distributed systems that support some forms of load balancing [10-13]. When inter-host communication cost is sufficiently low, response time and resource utilization of a distributed system can be improved further by partitioning jobs into tasks and assigning tasks to different hosts to be executed concurrently whenever possible. This approach is the same as the multitasking approach taken to increase parallelism in tightly-coupled multiprocessor systems. It is made feasible recently by the advent of extremely high-speed, local-area networks. Several optimal and suboptimal task assignment algorithms have been designed to find assignments of tasks to hosts in distributed systems [14,15]. These algorithms take into account inter-task communication times, which are often large compared with task execution times when tasks belonging to the same job are assigned to different hosts.

Load balancing and distributed task assignment schemes essentially extend the scheduling and resource management methods used in single host systems to support job sequencing in distributed environments. Typically, these schemes require the programmer to break up jobs into tasks and explicitly handle inter-task communication. Complexities of load balancing and task assignment algorithms grow with the number of tasks. These factors make it difficult to achieve a high degree of parallelism. When the applications supported by the distributed system require high reliability and availability, the need for fault tolerance and robustness makes load balancing and task assignment even more complex.

The problem of providing efficient, reliable services in distributed environments on highly available basis is made difficult partially by the fact that distributed systems are typically "Von Neumann machines." In such a system, executions of user processes are carried out under the

This research was supported in part by the U. S. Army under the contract No. DAAB07-84-K-K526 and by NASA under the contract No. NAG 1-613.

Authors' address: 1304 West Spring Avenue, Department of Computer Science, University of Illinois, Urbana, Illinois 61801

supervision of a system-wide controller that sequences user processes. To ensure that the controller is reliable and highly available, it is often necessary to distribute and/or replicate its functional modules on more than one host. The need to coordinate the operations of a distributed controller on different hosts and to reconfigure its structure in the presence of load fluctuations and failures is one of the causes of inefficiency. The large amount of state information that must be gathered and maintained by the controller to support its decisions is another cause.

One approach used to increase parallelism in a natural and robust manner in tightly-coupled multiprocessor environments is to use the dataflow architecture [16,17]. This paper proposes the use of this approach to increase parallelism, and to improve fault tolerance and availability in loosely-coupled distributed environments. A *macro dataflow model* is developed to model distributed computation. The macro dataflow model is based on the dataflow model, with features added to model interprocess communication and synchronization. A high-level, macro-dataflow, distributed system architecture based on this model is proposed. As in a dataflow machine, granules of computation in a macro-dataflow system are performed as soon as all necessary input are available. A high degree of parallelism can be achieved naturally without the need of a complex, distributed controller. Many fault tolerance techniques can be used in an integrated manner to ensure fault tolerance and to improve system availability. An object-oriented approach to software design and implementation can be used to support the development of macro dataflow programs.

The rest of the paper is organized as follows. Section II describes briefly the well-known dataflow model and motivates the macro dataflow model. The elements of macro dataflow model are presented in Section III. Section IV proposes an architecture of distributed, macro-dataflow systems for executing macro-dataflow programs. The advantages of macro-dataflow distributed system are also discussed. Section V is a summary.

II. Background and Motivation

In the well-known dataflow model, there are two types of objects: tokens and actors. Tokens carry data or control information. Each actor performs a function based on the information contained in the tokens it consumes. Actors are computation primitives, corresponding to granules of computation. In most cases, granules of computation are machine instructions [16,17]. Actors do not have any internal state preserved from one computation to the next.

A computation can be described by a dataflow graph in which nodes are actors. Actors are connected by directed arcs along which tokens flow. Arcs model data dependencies between granules of computation. Specifically, the execution of any granule cannot begin until all granules of computation on which it depends have completed. Correspondingly, in the model, an actor is enabled and may "fire" (execute) only when there are tokens on all of its incoming arcs. Parallelism is gained in dataflow architectures by allowing any actor to execute on any processor, and by allowing as many enabled actors to fire as there are processors to execute them. When there is a sufficiently large number of processors, only actors that depend on uncompleted granules of computation are not enabled.

One disadvantage of the dataflow approach is that the amount of inter-processor communication can be quite high, since each computation can require many tokens to be sent. In most distributed systems where inter-host communication is costly, the advantage in the high degree of parallelism gained by using this approach is lost unless lower resolution is used [17]. In other words, instead of individual machine instructions, more complex, high-level functions should be chosen as computation primitives. To illustrate this point, let us consider the dataflow graph

shown in Figure 1. The program represented by this graph generates the sum $1^2 + 2^2 + \dots + n^2$. The dataflow graph consists of three subgraphs. The first subgraph, consisting of actors D, E, and F, is a counter. This subgraph generates a token for each integer from 1 to n and sends the token to the second subgraph, consisting of the actor C. For each integer less than n , the first subgraph also generates a control token with the value of *true*. A false token is generated when the value n has been reached. The control tokens are sent to the third subgraph, consisting of actors A and B. The second subgraph receives inputs tokens containing integers from 1 to n and computes the squares of these integers. It passes the results on to the third subgraph that computes the sum of the squares. The sequential implementation of the algorithm implemented by this dataflow program is shown below.

	LOAD	X,0	Load 0 in register X
	LOAD	Z,0	Load 0 in register Z
LOOP:	INC	X	Increment X
	MOV	Y,X	Move contents of X to Y
	MULPY	Y,Y	Multiple Y to Y
	ADD	Z,Y	Add the content of Y to content of Z
	CMP	X,n	Compare the content of X with n
	JNE	LOOP	Jump to LOOP if X is less than n
DONE:			

Suppose that each instruction takes one time unit. When communication time is negligible, the sum of n squares is obtained in $2n+3$ time units for the dataflow implementation, and $6n+2$ time units for the sequential implementation. $10n$ tokens are passed between actors for the dataflow implementation while only two messages are required, one to send the input n and one to output the result, for the sequential implementation. Let t be the amount of time required to send a message or a token. Let C_d be and C_s be the completion time for the generation of the sum using the dataflow implementation and the sequential implementation, respectively. Assuming that any enabled actor fires immediately, and that all communication can be performed in parallel, we have that C_d is equal to $n(2t+2)+2t+3$ and C_s is equal to $6n+2t$. For $n=10$, C_d becomes larger than C_s when t becomes larger than 2 time units. In distributed systems, t being larger than 2 is almost always true.

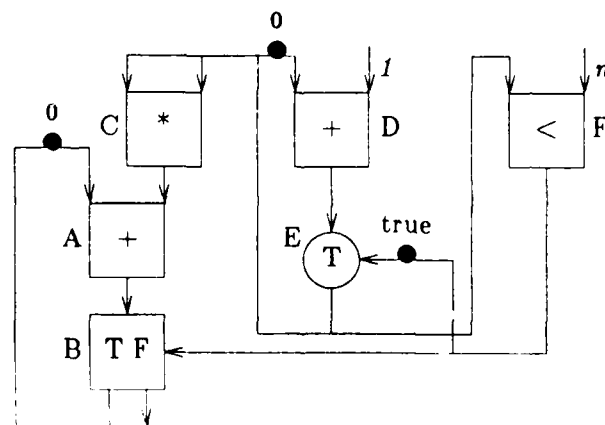


Figure 1. An Example of Dataflow Implementation

We refer to the average number of messages or tokens sent per computation primitive executed in a computation as the *communication ratio* of the computation. The larger the communication ratio of a computation is, the faster its completion time grows with the time required to send a message or token. Hence, to adapt the dataflow approach to a loosely-coupled system, where the time required to send a message or token is typically in the order of tens or hundreds of machines instruction time, it is necessary to reduce the communication ratio of computations. This can be done by increasing the power and complexity of the computation primitives, i.e., by increasing the size of the granules of computation.

III. Macro Data Flow Model

In the macro dataflow model of distributed computations, *macro actors* perform complex, high-level functions instead of individual machine instructions. For example, in a distributed relational database system, one may choose to let computation primitives be operations such as database-read, database-write, project, select, join, etc., required to process a transaction. Similarly, in a electronic message system, jobs are requests to send and receive messages. Macro actors may be edit-file, encrypt-file, send-file, write-mailbox, etc..

III.1. Regular Actors, Tokens, and Program Graphs

Some of the macro actors are *regular actors*. They are the same as actors in the dataflow model. (Unless stated otherwise, we use the term actors to mean macro actors in subsequent discussion.) (Regular macro) actors have the following characteristics:

- (1) All actors of a given type are equivalent.
- (2) Each type of actors requires a fixed number of input tokens, each of which must be of a specific type. When all required tokens are available, the actor is enabled.
- (3) An actor may execute only when enabled.
- (4) An actor performs some computation, generating output tokens that depend only on the input tokens.
- (5) An actor need not send the same output tokens to all its recipients.
- (6) A actor may have internal state during the course of a single execution but no state information is preserved from one execution to another.

Similar to a dataflow graph, a macro-dataflow graph is a high-level view of a program. Nodes in this graph are actors. There is an arc from the actor v to the actor u when there is data dependency between v and u . Tokens flow along the arcs between actors carrying both data and control information. Parallel execution of a macro dataflow program can be realized by firing each actor as it becomes enabled. It is not necessary to be concerned with synchronizing the execution of individual actors within a program because all allowed orders of execution of actors vis a vis each other are specified by the arcs in the macro dataflow graph.

III.2. Persistent Actors

The macro dataflow model described so far is a straight forward extension of the dataflow model. It makes no provision for the sharing of information between programs, or side effects of any program on other programs. All information transfers occur through the use of token flow. Hence, there is no transfer of information between dataflow programs unless there are arcs connecting actors in the corresponding dataflow graphs. These connecting arcs effectively make the programs into a larger program. In general, it is difficult to model interprocess communication and global serialization required by many applications using this scheme. This scheme presumes

the knowledge of other programs, their relationship to each other, and the order of their executions. This knowledge is often not possible. For example, suppose that the programs are transactions to a database. Information is transferred between transactions since they access the same database. It is necessary to serialize the executions of transactions that read and write the same data items in order to maintain database consistency. But it is not possible to construct a composite dataflow graph modeling interleaved executions of all possible transactions to the database allowed by the database manager.

To model communication between programs and side effects of programs, we introduce the notions of persistent data and *persistent actors*. Persistent actors have the same characteristics as regular actors except (1), (4), and (6) discussed above. A persistent actor maintains state information that is preserved from one firing to the next. Hence, the output tokens generated by a persistent actor for different firings are not necessarily the same for the same input tokens. Since each instance of a particular persistent actor type can have a different internal state, different instances are not identical. We note that the notion of persistent actors is very similar to the concept of monitors in Concurrent Pascal and objects in object-oriented systems. By adding persistent actors to the macro dataflow model, the arcs between actors in a program graph no longer completely specify all data dependencies between all granules of computation carried out by the system. In particular, persistent actors provide us with a way to model information transfer between actors in different programs as well as within a program. We propose to use the dataflow graph of each program to completely specify data dependency relationship between actors within each program and limit the use of persistent actors to model synchronization and serialization between different programs.

To illustrate the use of persistent actors, we consider a database system as an example. Suppose that every transaction (i.e., a program) in this system is represented by a macro dataflow graph containing four actors: database-read, select, project, and database-write. Suppose that in the macro dataflow graph, these actors are connected as a directed path; select and project are enabled when database-read is completed and database-write is enabled when select and project are both completed. This graph models the dependency between operations within each transaction. However, it does not model the interference between operations of different transactions. To model this type of interaction between programs, we let select and project be regular actors and database-read and database-write be persistent actors. To be specific, suppose that a server module called database manager is provided by the system to execute the actors database-read and database-write. The database manager enforces some form of concurrency control policy. As a result, the value returned by the database manager after serving a database-read request may be data retrieved on the behalf of the transaction or a "database-read denied" message. Similarly, a database-write operation may be successively carried out or failed depending on the operations of other transactions. This type of interactions can be modeled by having the internal state of persistent actors preserved from one firing to another. The output produced by persistent actors database-read and database-write depend not only on their input tokens, but also on the sequence of read and write operations already carried out by the database manager, that is, the previous firing of these actors.

When a database-read request is denied, subsequent operations (i.e., select, project, and database-write in our example) are aborted and some appropriate recovery action is then taken in typically database systems. One way to model aborted operations is to let each firing of a persistent actor have at least two possible outcomes. The first outcome is "success". In this case, the output tokens generated by the persistent actor are sent to enable actors connected to it in the subgraph representing the program for the transaction. We call the portion of the computation

following the persistent actor its success continuation. The second outcome is "failure" (e.g., due to a synchronization error, or a data out of bounds.) In this case, output tokens are directed to actors which perform whatever operations are necessary, (e.g., rollback, re-try, notify user, etc..) For example, these actors may be in the subgraph representing the program for rollback and recovery functions. We call this the failure continuation of the persistent actor. In general, other continuations are also possible. There is no need to specify what they are here.

III.3. Macro Data Flow Program Composition

A macro data-flow program specifies the actors performing the desired computation primitives and the graph describing the dependency between the actors. To produce such a program, a programmer may take the traditional approach to let a compiler determine the parallelism inherent in his source code (such as calls to predefined procedures or code segments without data dependencies) and generate the actors and the program graph. Alternatively, programmer may use any existing types of actors (i.e., program modules capable of carrying out the corresponding computation primitives,) or write new types of actors to provide functionalities not supported by any existing type. The new types, once written and defined, become existing types and can be used later in other macro dataflow programs. The entire macro dataflow program would be represented as a hierarchical graph constructed from actors and predefined subgraphs. Thus, actor types are reusable program segments and the macro dataflow model may be used to support software reuse.

We note that an object-oriented approach to software design and implementation provides the ideal support to the design and development of macro dataflow programs. Indeed, macro actor types are functional objects. These objects can be instantiated to provide the required functionalities. The arcs carrying tokens in the macro dataflow graphs are also objects. These objects are instantiated to provide a communication facility between the functional objects. Objects can be easily implemented in programming languages such as Ada which provide packages to encapsulate objects, hide implementation details, and allow parametrized interfaces.

In general, the process of choosing the functionality and complexity of the computation primitives, and hence actor types or functional objects, is a software system design process. The ideal choices of computation primitives are application dependent. For applications where good response and high throughput are essential, the computation primitives should be chosen to achieve near optimal trade-off between low communication overhead and a high degree of parallelism. For applications where fault tolerance and availability are essential, computation primitives should be chosen so that they correspond to atomic actions that can be carried out reliably and supported on a highly available basis.

IV. Macro Dataflow Distributed Architecture

We propose to support the macro dataflow model on a distributed system in which entities capable of executing computation primitives are server modules. Each host may choose to provide a set of server modules to support frequently required, well defined functionalities on the behalf of user. Examples of server modules include those which perform sort, merge, Ada-compile, etc. when supplied with input data. These modules have attributes of library routines; they perform functions based on inputs arguments or manipulate data structures. Other examples of server modules include database managers and file servers. These server modules have attributes of processes and may maintain persistent data. In general, hosts may provide server modules such as terminal handlers, message handlers, voice and video image mailers, image processors, printer servers, text processors, language compilers, communication servers, computation servers, file

servers, etc.. Jointly, these server modules support a comprehensive set of communication and computation primitives, isolating the user from the boundaries of host systems and networks, as well as from differences in operating systems.

Even though the macro dataflow programs are very similar to dataflow programs except for the size and complexity of computation primitives, the architectural support required by the macro dataflow computation is very different from that of the dataflow machines. In a macro dataflow system, the execution of each computation primitive, corresponding to a client process to some server module, is carried out whenever all the necessary input data and control information are available to the module, i.e., in parallel whenever possible. The need to have a system-wide controller for scheduling and synchronizing task execution purposes is eliminated. Thus, the overall system control structure can be simplified and its robustness improved. Server modules that can be instantiated and activated to execute computation primitives on each host are computational resources on the host. These server modules play the same role in a macro dataflow system as the roll played by hardware resources capable of carrying out machine instructions in dataflow machines. However, unlike hardware resources, server modules can be dynamically instantiated on any host, and once instantiated, can be destroyed or migrated to different hosts. For this reason, resource management in a macro dataflow system is more complex than in dataflow machines, where hardware resources are often allocated to computations statically. When a computation primitive in a program is to be carried out, the resource manager in a macro dataflow system must identify the type of server modules capable of carrying out the primitive, locate or instantiate one, and assign it to the program. A distributed name server should be used to provide naming and directory services to the resource manager, in this case. (This name server should not be more complex than the name servers required to support location-transparent, interprocess communication in traditional distributed systems, since much of the same functionalities are required in both cases.) To support naming and directory services will need precise functional specifications and interfaces specifications of server module types

A concept that may provide a general framework for the specifications of server module types is the concept of virtual services and virtual service protocols. A virtual service protocol is a presentation-layer protocol compatible with the ISO Open System Interconnection (OSI) reference model. It provides a structured approach for characterizing and defining a (virtual) service provided by a type of server modules, as well as for specifying the interactions between a server module providing the service and a client using the service. In other words, a virtual service protocol defines a generic service and presents a standardized interface to server modules providing the service. This interface can be parametrized using the concept of option negotiation first introduced in virtual terminal protocols. (Stated in terms used in software specification, a virtual service protocol is equivalent to the functional and interfaces specifications of server module type.) Virtual service protocols generalize the well known notions of virtual terminal protocol, file transfer protocol, and virtual graphics protocol, which are used to support terminal handling, file transfer, and graphics, respectively, to provide a coherent set of utilities in a distributed system.

V. Summary

The macro dataflow model for distributed computation is presented. Many different models of distributed and concurrent computations such as remote procedure calls, parbegin-end constructs, monitors, and objects can be easily represented in this model. It is patterned after the dataflow model, but with two important differences. First, the granularity of the computation primitives in the macro dataflow model is arbitrarily complex. Using more computationally complex actors reduces the communication ratio of macro dataflow programs, and makes the

underlying network less of a bottleneck. Second, the notion of the persistent actors is introduced to model inter-program communication and side effects of programs. Persistent actors also provide a convenient way to model abstract data objects, servers, and other constructs that are awkward to model in the dataflow model. Many questions remain open. One open question is how to model fault-tolerant computation in the macro dataflow model. Another question is whether secure access control and information flow control mechanisms can be more easily designed in a macro dataflow system.

This paper proposes a distributed system architecture based on the macro dataflow model of computation. This architecture appears to provide a framework within which high degrees of parallelism, fault tolerance, and availability can be achieved naturally in a distributed environment. An important problem to be solved concerns with the provision of necessary underlying support to the resource management function. Naming, location, and selection of server modules to perform computation primitives invoked in programs, and the use of techniques such as atomic remote procedure calls and resilient procedures to ensure fault tolerance and improve availability are some of the issues to be addressed.

References

- [1] Chow, Y. C. and W. H. Kohler, "Models of dynamic load balancing in heterogeneous multiple processor systems," *IEEE Transactions on Computers*, Vol. C-28, No. 5, pp.354-361, 1979.
- [2] Stankovic, J. A., "The analysis of a decentralized control algorithm for job scheduling utilizing Bayesian decision theory," *Proceedings of the 1981 International Conference on Parallel Processing*, 1981.
- [3] Kwang, K. et.al, "A Unix-based local computer network with load balancing," *Computer*, April, 1982.
- [4] Chou, T. C. K. and J. A. Abraham, "Load balancing in distributed systems," *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 4, July, 1982.
- [5] Gao, C, J. W. S. Liu, and M. R. Railey, "Load balancing algorithms in homogeneous distributed systems," *Proceedings of 1984 International Conference on Parallel Processing*, August, 1984.
- [6] Wang, Y. T. and R. J. T. Morris, "Load sharing in distributed systems," *IEEE Transactions on Computers*, Vol. C-34, No. 3, March, 1985.
- [7] Tantawi, A. N. and D. Towsley, "Optimal static load balancing in distributed computer systems," *Journal of the ACM*, Vol. 32, No.2, April, 1985.
- [8] Eager, D. L., E. D. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 5, May, 1986.
- [9] Hsu, C. Y. H. and J. W. S. Liu, "Dynamic load balancing algorithms in homogeneous distributed systems," *Proceedings of the 6th International Conference on Distributed Computing Systems*, May, 1986.
- [10] Powell, M. L. and B. P. Miller, "Process migration in DEMOS/MP," *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, 1983.
- [11] Popek, G. et al., "LOCUS: A network transparent, high reliability distributed system," *Proceedings of the 8th Symposium on Operating System Principles*, pp. 169-177, December 1981.
- [12] Rashid, R. F, and G. G. Robertson, "Accent: a communication oriented network operating system kernel", *Proceedings of the 8th Symposium on Operating System Principles*, pp. 64-75, December 1981.
- [13] Theimer, M. M., K. A. Lantz, and D. R. Cheriton, "Preemptable remote execution facilities for the V-System," *ACM Operating Systems Review*, Vol. 19, No.5 and *Proceedings of the Tenth ACM Symposium on Operating System Principles*, December 198
- [14] Stone, H. S., "Multiprocessor scheduling with the aid of network flow algorithms," *IEEE Transactions on Software Engineering*, Vol SE-3, No. 1, 1977.
- [15] Lo, V., "Task assignment in distributed multiprocessor systems," Technical Report No. UIUCDCS-R-83-1144, Department of Computer Science, University of Illinois at Urbana-Champaign, Illinois, in a distributed computer system,
- [16] Srin, V. P., "An architectural comparison of dataflow systems," *Computer*, March, 1986.
- [17] Gaudiot, J. L. and M. D. Ercegovic, "Performance Analysis of a Data-Flow Computer with Variable Resolution Actors," *Proceeding of 1984 IEEE Conference on Distributed Systems*, 1984

Kemal Efe

Computer Science Department
University of Missouri-Columbia

ABSTRACT

There are many heuristic algorithms in the literature for task assignment in Distributed Computing Systems. A common goal in all of these algorithms is to minimize the communication cost of an assignment. Each of the researchers who develop heuristic algorithms claim that their algorithms perform very well. The purpose of this paper is to compare the performance of four algorithms in the literature in terms of both accuracy and algorithmic efficiency. It is concluded that the trade-off between optimal but slow algorithms v.s. sub-optimal but efficient ones heavily favors the heuristic approach. A heuristic algorithm is derived to combine the most successful parts of the reviewed papers, and some remaining important problems are identified.

INTRODUCTION

We loosely define a distributed computing system as a combination of two or more computers, each with private memory and which works under the control of a two-layer operating system: The lower layer consists of the collection of local operating systems that control the local activities of individual computers, and the higher layer is a network-wide operating system which coordinates the activities involved in resource and load sharing. Each of these layers may contain several sublayers as suggested in the 7-layer ISO model [11], but for the purposes of this paper these sublayers are transparent. What is important, however, is that, the communication between computers occurs over some communication medium with a bandwidth several times lower than that between a processor and its local memory.

An important research area in the design of the higher level operating system is development of efficient algorithms for load sharing. The importance of load sharing can be attributed to the expected level of power improvement that would be attained. A measure of power of any computer system can be defined as the ratio T/R , where T is the throughput and R is the average response time of the system. To maximize power we need to maximize throughput while minimizing average response time. Without load sharing, the total throughput of a system of P processors will heavily depend on the distribution of service requests arriving at individual processors. In this sense, the purpose of load sharing is to re-distribute service requests uniformly over the set of available processors, so that the sensitivity of the power of a system to distribution of service request arrivals at different sites may be minimized. In general, we expect the distribution of job ar-

rivals to individual sites to be variable. Therefore, dynamic algorithms are needed for real-time assignment of jobs to processors.

In the literature both deterministic and probabilistic models have been successfully used in the area of multiprocessor scheduling, where communication delays between processors are negligible (either due to shared memory or high bandwidth communication medium). However these models can not be extended to distributed processing where communication delays are high. Recently, load balancing algorithms were proposed for distributed computing systems based on some form of "bidding" or "drafting" protocol [9,10], but these algorithms are only applicable for the assignment of independent tasks, or when a task has been partitioned into its constituent processes before allocation starts.

We are particularly interested in the problem of task partitioning which minimizes the communication costs subject to constraints such as real-time, resource availability, or any other constraints that may be desirable for a particular application. Such a problem has been formulated as an integer programming problem by a number of researchers. However, since the problem is NP-complete, good heuristic algorithms have received due attention in the literature. The purpose of this paper is to compare the performance of a number of heuristic algorithms proposed by different authors. For this comparison we give a mathematical programming model of the problem first. We then describe the heuristic algorithms proposed for the approximate solution of this problem. Since no two models are exactly the same, a direct comparison is not possible. Therefore we had to make small adaptations so that a uniform common ground may be derived, and a single set of terminology may be used without changing the basic algorithm.

Problem Formulation

We assume that the processors in a distributed system are fully interconnected. A task consists of a number of modules that may communicate with each other, and is characterized by two matrices: (a) A cost matrix $[t(i,j)]$ which indicates the execution cost of module i on processor j , and, (b) A communication matrix $[c(i,k)]$ which represents the cost of communication between modules i and k . If two modules i and k do not communicate with each other then $c(i,k)=0$, and $c(i,k)>0$ otherwise. A decision matrix $[x(i,j)]$ is defined which represents an assignment such that $x(i,j)=1$ if module i is assigned to processor j , and $x(i,j)=0$ otherwise. The total execution cost of an assignment is expressed as:

$$\sum_i \sum_j t(i,j) x(i,j)$$

If the computers are homogeneous this term is eliminated from the problem formulation. The total communication cost of an assignment incurred between modules not assigned to the same processor is:

$$\sum_i \sum_j \sum_{k>i} c(i,k)(1-x(i,j))x(k,j)$$

Hence the total cost of an assignment is the sum of communication and processing costs:

$$\text{COST} = \sum_i \sum_j (t(i,j)x(i,j) + \sum_{k>i} c(i,k)(1-x(i,j))x(k,j))$$

The optimal assignment is one which minimizes this cost function subject to various constraints that may be specified. Table 1 shows the constraints included in the models which we consider in this paper. In all of these models, minimization of the communication cost between communicating processes is considered to be the primary goal of task assignment. All of these models (but one) assume that processors are homogeneous, and the cost of communication depends solely on the volume of data (distance factor is ignored). Other considerations between various models include: different processor speeds [3], processors with unique resources [2,5], real-time constraints [2,4,8], and load balancing [2].

MODEL (ALG)	MINIMIZE COMM	PROC SPEED	DIST'N NODES	REAL-TIME CONSTRAINT	LOAD BAL
HJ	O(V ⁴)	same	yes	no	no
GE	O(VE)	same	no	yes	no
E	O(VE)	same	yes	yes	yes
L	O(V ⁴)	different	yes	no	no

Table 1

As we see in table 1, no two models are exactly same. Therefore a direct comparison is not possible. In the rest of this paper we first describe how these algorithms try to minimize the communication costs in an assignment and compare them in terms of running time complexity. We then describe how various algorithms are extended to handle the real-time constraints, and compare their rate of success in handling the real-time constraint as described by each model. We close this discussion by giving the results of this research and discussing further problems for future research.

II ALGORITHMS FOR MINIMIZATION OF COMMUNICATION COSTS:

A task to be partitioned is represented by an undirected graph in which nodes represent the program modules and arcs represent the communication between them. Nodes and arcs are labeled by numbers that represent the cost of module execution and communication, respectively. Also character labels are associated with nodes for convenience in referring to a node in discussions below. A partitioning of the graph that represents a task designates the allocation of modules represented by the nodes in the graph, as each subgraph obtained after partitioning represents a cluster of nodes to be assigned to the same processor. The

communication cost of an assignment equals to the total communication between these clusters since we assume that the communication cost between the nodes within a cluster is zero. In the rest of this paper we refer to each algorithm by the initials of its author.

Algorithm HG

One of the earlier algorithms developed for minimizing the communication cost of an assignment is due to K. Haessig and C.J. Jenny [5]. In this model it is assumed that some of the nodes, called "distinguished nodes," have some unique resource requirements and that there is a one-to-one correspondence between distinguished nodes and processor locations. Therefore the assignment of distinguished nodes are fixed. The rest of nodes are subject to assignment for minimization of communication costs. A partitioning is said to be "feasible" if there is exactly one distinguished node in each of the clusters obtained by the assignment.

The algorithm consists of two phases: In the first phase a feasible solution is obtained which is then improved in the second phase. For the determination of a feasible solution the process graph is first extended as follows: Assume in the process graph of figure 1.a the nodes V1 and V2 are distinguished nodes. We introduce an additional node, V7 in this case, and connect this node to the distinguished nodes V1 and V2 with arc weights larger than any other in the process graph. After this extension, we find the maximum spanning tree of this graph as shown in figure 1.b. Now if we drop the node V7 and its outgoing arcs we obtain a disconnected graph which contains all of the nodes of the original process graph. This partitioning of nodes forms a feasible solution and a first approximation to the optimum cut-set.

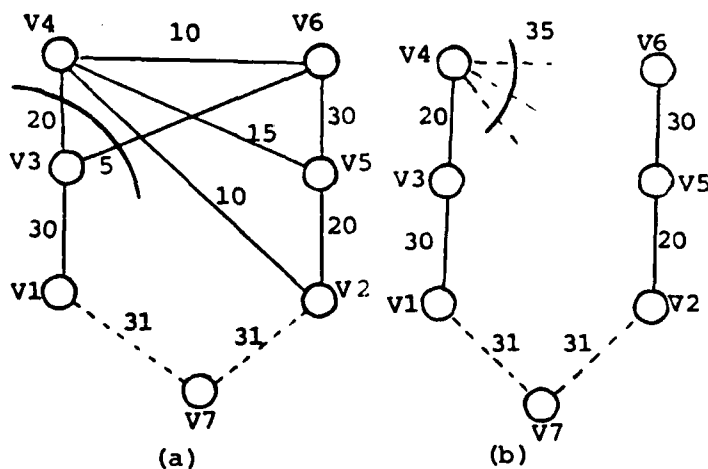


Figure 1: Phase 1 of Algorithm HG.
(a) Process graph, (b) Maximum spanning tree.

In the second phase some nodes are relocated from one subgraph to another if it "pays" to do so. To describe the proposed algorithm for the relocation of nodes, assume a graph is partitioned into two subgraphs as shown in figure 2. Here CUT-1 represents the partitioning found at the end of the first phase and A represents the cost of this partitioning. We now take the sub-graphs one at a time and check if a cut with lower communication exists between the distinguished node and another node in that cluster. If a new cut, say CUT-1.a is found with cost B such that $B < A$ then this cut is preferred over CUT-1. This process is repeated for every cluster, reassigning some nodes more than once if necessary (e.g. if a cut is found with cost C such that $C < B$).

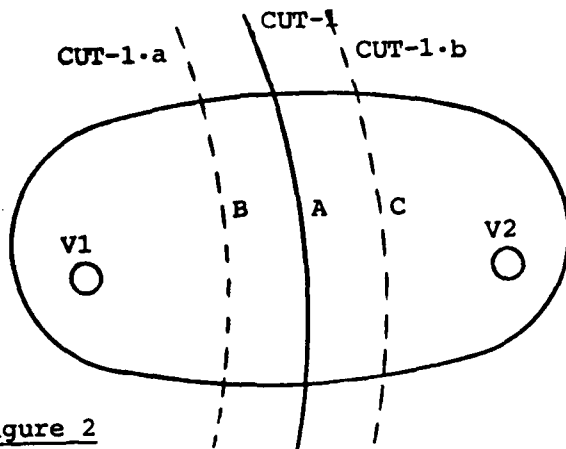


Figure 2

To determine whether such a cut exists, the authors propose to use a min-cut max-flow algorithm between the distinguished node and other nodes in the same cluster. Although the authors propose ways to minimize the number of attempts to find such reassignments, in the worst case we attempt as many times as there are nodes in the cluster.

The most time consuming part of this algorithm is the second phase where a min-cut max-flow algorithm is used. In the worst case, for a graph with V nodes P of which are distinguished, the first phase may assign just the distinguished nodes in the first $(p-1)$ clusters and the remaining $(V-P+1)$ nodes to the P th cluster. For an reassignment to be made, we may need to repeat the min-cut max-flow algorithm $(V-P)$ times on a graph with $(V-P+1)$ nodes. Since the running time complexity of the min-cut max-flow algorithm is a third order function of the number of nodes [1], for the above algorithm we have a worst case estimation of $O((V-P)(V-P+1)^3)$, or if $P \ll V$, we have $O(V^4)$.

Algorithm GE

GYlys and Edwards [4] proposed a much simpler algorithm than the one above. The basic idea is again to form clusters of nodes in such a way that the total inter-cluster communication is minimum. Initially we start with V clusters such that there is a cluster for each node and that no node is contained in more than one cluster. Each cluster is represented by a node in the process graph. For a partitioning between P processors the algorithm can be simply described as:

1. Repeat steps 1.a and 1.b until number of nodes is reduced to P

1.a Find the cluster pair with maximum communication

1.b Fuse them

The process of fusion in step 1.b involves modifying the process graph so that the two clusters are now represented by a single node (see figure.3), the arcs from the new node to others represent the combined communication of nodes in the cluster.

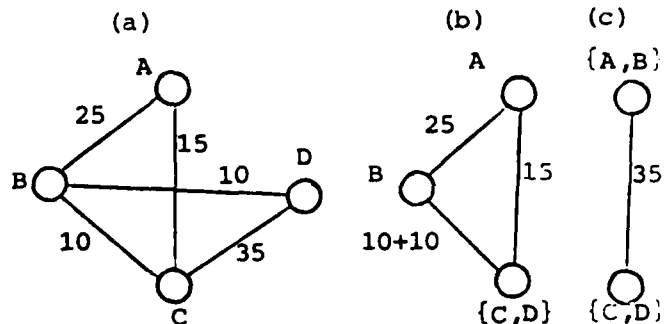


Figure 3: (a) Process graph, (b) after fusing C and D. (c) After fusing A and B.

This algorithm necessitates finding the arc with minimum weight at each iteration. If there are E arcs in the process graph, $O(E)$ will be the search complexity by using a linear search algorithm. Since this process will be repeated $(V-P)$ times, the algorithmic complexity will be $O((V-P)E)$, or if $P \ll V$ we have $O(VE)$.

The authors also report the results from other clustering techniques based on some measure of association between modules. They experimented with many different measures in the literature, and found that the success of these algorithms heavily depends on the "right" choice of initial cluster centroids. Due to the lack of good algorithms to determine the "right" choice in a process graph, we are unable to pursue this option further at this stage.

Algorithm E

In another algorithm proposed by K. Efe [2], instead of searching for the maximum of all arc weights, we search for a node pair which communicate with each other more than they communicate with the rest of the nodes in the process graph. More formally, we search for a node pair k and l such that

$$\begin{aligned} c(k,l) &> c(k,i); & i=1..V, i \neq l \\ c(k,l) &> c(l,i); & i=1..V, i \neq k \end{aligned}$$

To find such a pair, the algorithm first subdivides the set of all arcs $S(E)$ into V sets of $S(i)$, where $S(i)$ is the set of arcs emanating from node i . The algorithm proceeds as follows: (The reader should easily follow the steps of this algorithm by considering the figure-4)

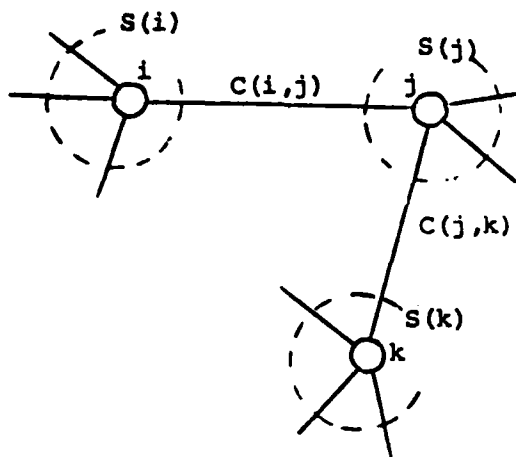


Figure 4

1. Pick a node i at random
2. Find j such that $c(i,j) = \text{MAX}\{S(i)\}$
3. Find k such that $c(j,k) = \text{MAX}\{S(j)\}$
4. If $i=k$ then execute 4.a, else execute 4.b.
 - 4.a Fuse i and j and assign label i to the new node obtained after this fusion. Set $V \leftarrow V-1$; if $V=P$ then stop, else go to step-2.
 - 4.b Set $i \leftarrow j$; $j \leftarrow k$; go to step-3.

The more general version of this algorithm described in 2 also considers the notion of distinguished nodes. The modification required for partitioning with distinguished nodes in the graph is rather simple: When a pair of nodes are found that satisfies the inequality above, we check if they are both distinguished. If at least one of the nodes is a non-distinguished node then we fuse them, otherwise the arc between them is deleted and search continues from the node i .

In steps (2) or (3) of this algorithm a search is made among S^* arcs, where $S^* = \{S(i); i=1..V\}$. In the worst case we may walk through V nodes, searching among the $2E$ arcs contained in all sets of $S(i); i=1..V$. Hence the total number of elements will be $2E$ for each fusion, and $O((V-P)2E)$ for the entire algorithm. Comparing to the algorithm GE this order of complexity seems to be slower by a factor of two. However, simulation results show that algorithm E is indeed marginally faster than algorithm GE because of the way the search is made. Notice that the running time complexity of algorithm GE is exact, while in algorithm E the search stops without necessarily searching all possibilities.

Algorithm L

Another model that is based on min-cut max-flow algorithm is proposed by V.M. Lo [8]. In this model processor speeds are assumed to be different. A system of V modules and P processors is modeled as a graph in which each processor is a distinguished node (see figure-5). An arc is drawn from each process node i to each distinguished node q with the weight

Execution Costs				Communication Costs					
	P1	P2	P3		t1	t2	t3	t4	t5
t1	3	11	23	t1	0	15	2	0	0
t2	1	100	9	t2	15	0	0	0	0
t3	14	10	21	t3	2	0	0	12	17
t4	15	6	8	t4	0	0	12	0	23
t5	100	2	7	t5	0	0	17	23	0

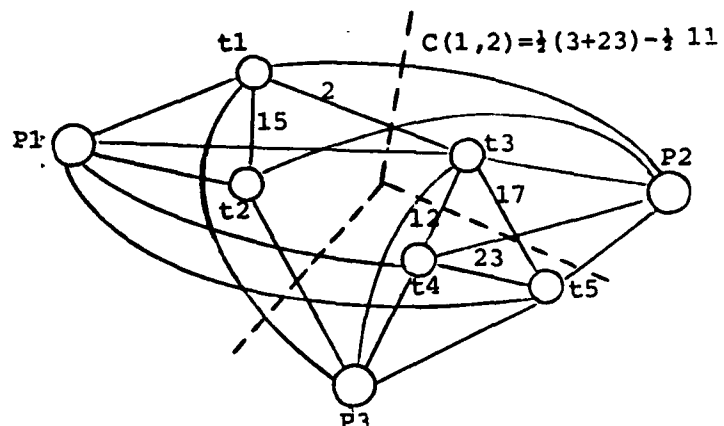


Figure 5

$$c(i,q) = \frac{1}{P-1} \sum_{r \neq q} t(i,r) - \frac{P-2}{P-1} t(i,q)$$

where $t(i,r)$ is the cost of executing module i on processor r . If the processors are homogeneous, that is, if $t(i,r)=t(i)$ for $r=1..P$, then this equation reduces to $c(i,q)=t(i)/(P-1)$. In a comparison with the algorithms above it is more appropriate to use this reduced form of $c(i,q)$ to make the modelling assumptions similar.

As in the algorithm HJ, a feasible partitioning contains exactly one distinguished node in each cluster. If there are only two distinguished nodes a min-cut max-flow algorithm guarantees the optimum result. For more than two processors, the author suggests a three phase heuristic algorithm as follows:

1. Repeat phase 1 (below) until assignment is complete or no assignment is made in the last iteration.
2. If there are nodes remaining unassigned then try to make a "lump" assignment as described below (phase 2).
3. If a "lump" assignment can not be made then use a greedy assignment algorithm as described in phase 3 below.

Phase 1 To force a cut between a processor node k and the rest of the graph, let P be the set of processor nodes, and let $k'=P-\{k\}$.

1. For $k=1..P$ repeat steps 1.a and 1.b.
 - 1.a Fuse all nodes in k' into a single node
 - 1.b Run min-cut max-flow algorithm between k and k'

2. If there are no nodes remaining unassigned, or if no assignment has been made in step 1 then stop else execute step 3

3. Recalculate the arc weight increments for processor nodes (to be added on the communications of nodes already assigned) on the reduced graph of remaining nodes and go to step 1.

Phase 2 In this phase a lower bound L is calculated on P -way assignment by running the min-cut max-flow algorithm between any arbitrarily chosen pair of processor nodes. If L is greater than the cost of assigning all remaining nodes to one of the processors then all remaining nodes are assigned to one processor. Otherwise phase 3 is activated.

Phase 3 In this phase the average communication cost C between all remaining nodes is calculated. Then the arcs with a weight less than or equal to C are dropped. After this step we locate clusters of nodes between which communication costs are "large". Each cluster is then assigned to the processor where it will incur minimum communication cost.

The most important determinant of the running time of this algorithm is the phase 1, where a min-cut max-flow algorithm is used. In a graph with V nodes, a partitioning between P processors will require running this algorithm for a graph of $V+2$ nodes. Thus, for each iteration of the min-cut max-flow algorithm we have the running time $O((V+2)^3)$. In the worst case the algorithm may assign exactly one node to a processor at each iteration, completing an optimum assignment in V iterations. The worst case running time will then be $O(V(V+2)^3)$, which is roughly $O(V^4)$.

MODELS FOR REAL-TIME CONSTRAINTS

Three of the four algorithms we described above provide mechanisms for observing a real-time constraint. Below we describe these algorithms.

Algorithm GE

The real-time constraint in algorithm GE is incorporated by a small modification of the one described above. Before the algorithm starts we define a preset upper bound L on the total running time of modules assigned to each processor. In the algorithm, before a fusion is made it is checked if a single processor can handle the two clusters. If the total running time of nodes in the two clusters is less than or equal to L then they are fused into a single cluster. Otherwise the arc between these two nodes is deleted and a new search is begun. The algorithm terminates when no more fusions can be made.

Algorithm E

In this algorithm we first run the heuristics for minimization of communication cost as the first phase. If the result does not satisfy the real-time constraint then we activate the second phase which is described below.

1. Calculate the average load level L for the processors. Then modify the original process graph as follows:

1.a For any processor with load level within a tolerance range $L \pm T$ delete the modules assigned to that processor.

1.b If a processor has a load less than $L - T$ then represent all of the modules in that processor by a single node.

1.c If a processor has a load level that is greater than $L + T$ then retain all of the nodes representing modules assigned to it.

2. Modify the arc weight for the nodes in category (1.b) so that arc weights are incremented proportional to the difference between processors with light load and processors with heavy load.

3. Using the algorithm E reassign some nodes to processors with light load until the total processing time in these processors reach the range $L \pm T$.

This algorithm is persistent in the sense that it is designed to satisfy the real-time constraint at any cost. A slightly different version reported in 2 will loosen the real-time constraint if the cost is going to be too high.

Algorithm L

The algorithm L described above does not provide any facilities to meet a real-time constraint. Instead, for the real-time constraint the author describes another algorithm based on a combination of phase (3) of algorithm L and a graph matching algorithm. However, in this model the real-time constraint is specified in terms of an upper bound on the number of modules at a processor. Due to this difference in the specification of real-time constraint we are unable to make a direct comparison. However, what is worth noting here is that, according to the author the algorithm found the optimum result in 82.9% of the tested cases.

IV COMPARISON OF ALGORITHMS

We have seen that algorithms HJ and L have running time complexities of roughly $O(V^4)$. Between these two, algorithm L seems to perform better, because the author is able to prove that if the assignment is complete at the end of phase 1 or 2 then it is optimum. This is a noteworthy point since it imposes a certain degree of confidence in the performance of the algorithm. On the other hand, one important fact remains: since the problem being solved does not exactly represent the real world, it is probably just as appropriate to consider a less accurate solution if it means a substantial saving in the running time complexity. The algorithms GE and E have a lower order of complexity in running time, and thus constitute reasonable alternatives for algorithm L. The following two theorems provide additional support for this argument in terms of accuracy under certain assumptions.

Theorem 1: If the process graph is a tree then algorithm GE minimizes the communication cost.

Proof: The proof directly follows from the consideration of the fact that a tree becomes disconnected by removing any of its arcs. For an optimum P -way partitioning we just need to remove those $P-1$ arcs that have the minimum weight. Algorithm GE essentially produces this effect by fusing the

node pair with maximum communication at each iteration.QED.

Theorem 2: If the process graph is a tree with P distinguished nodes in it, then algorithm E finds the minimum communication partitioning that separates the distinguished nodes. The proof is given in [3].

As a corollary result of theorem-2 it is easy to see that algorithm GE also finds the minimum communication partitioning for a tree with distinguished nodes in it. The significance of these theorems becomes apparent when we consider the fact that, to detect whether a process graph is a tree, all we need is to check if $E=V-1$. In such a case much simpler algorithms can be derived as implied by the theorem-1. Below we describe the simulation results obtained from comparing algorithms GE and E.

Simulation

For simulation, we implemented algorithms GE and E in Pascal and tested their performance using randomly generated graphs in which the number of nodes, V, varied between 10 and 100. The number of processors were between 2 and 20 (or $V/3$, whichever is smaller). Altogether we used 250 random graphs. Concerning the accuracy of results, without the real-time constraints, we observed no significant difference between the two algorithms. For 76% of the cases the results were either optimum or within less than 20% difference of the optimum. More details of these simulations are given in [3].

The running times are measured on Amdahl 580. Table-2 gives a typical running time measurement for the two algorithms. Each entry in this table shows the running time for a total of 25 problems with a given number of nodes. Algorithm E is marginally faster than GE due to the fact that a node obtained after fusion is very likely to have an arc with "large" weight. In such a case, the search time for the next iteration is minimized.

# OF NODES	RUN-TIME (MILLISECONDS / 25 PROBLEMS)	
	Algorithm GE	Algorithm E
10	152	144
20	483	397
30	995	880
40	1470	1125
60	4763	3960
80	6402	5621
100	9175	7475

Table 2

We also compared the success of the two algorithms in meeting a real-time constraint. We set an upper bound L on the completion of execution as

$$L = L(\text{ave}) + T$$

where $L(\text{ave})$ is the average running time over P processors and T is the tolerance calculated as 20% of $L(\text{ave})$. With this upper bound, an assignment is "successful" if all of the modules are assigned to the given P processors. Algorithm GE performed much worse than we expected. For large graphs (i.e. $V > 20$) it failed almost always. In a set of 25 graphs each with 10 nodes, only 7 cases

produced a successful assignment. Figure 6 shows a typical case when this algorithm fails.

Algorithm E produced successful assignments in 72% of the tested cases. When it did not succeed, it was because the process graph was disconnected after step-1 of the modification algorithm. Also for successful assignments a certain cost increase has been observed due to reassignments. This cost increase was generally less than 50% for $V < 40$, while up to 100% increases were observed for larger problems.

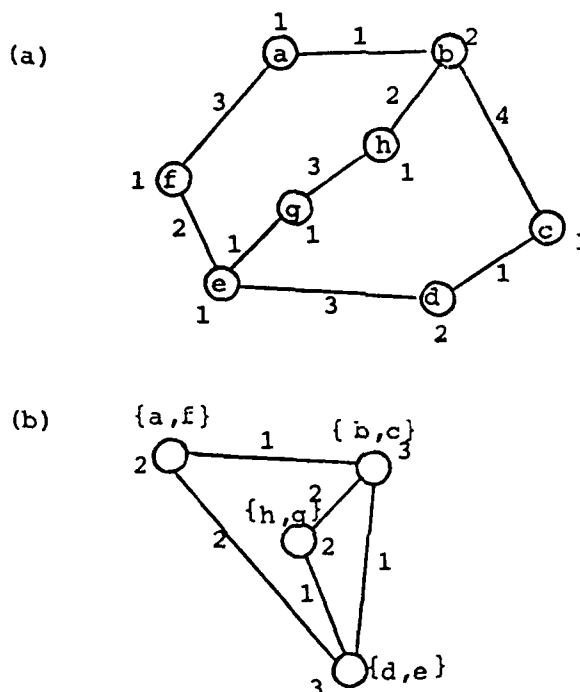


Figure 6: (a) Process graph, (b) After four fusions failure occurs if $P = 3$ and $L = 4$.

V CONCLUSIONS AND FUTURE DIRECTIONS

This research shows that the trade-off between using optimum but slow algorithms as opposed to sub-optimum but efficient ones heavily favors the heuristics when one considers on-line assignment of tasks to distributed processors. This argument is strengthened partly due to the success of heuristic algorithms in finding near optimal solutions, and partly by the fact that an optimal model is nothing more than an approximation of the real world.

As we have seen in this paper, different algorithms have their strengths and weaknesses. Taking algorithmic efficiency as the paramount goal, the following strategy can be adapted in designing an algorithm to combine the best ideas in the papers we considered:

1. Use algorithm GE or E for minimizing communication costs
2. Use algorithm E for real-time constraint
3. Use algorithm L if processor speeds are different.

Nevertheless, this algorithm will only provide a sub-optimal solution to a rather small proportion of the problems in task assignment. One important problem that remains is the consideration of precedence constraints between modules. Also on-line calculation of the cost of module execution and communication are open problems, although some limited success has been reported [6,7].

ACKNOWLEDGEMENTS

This research was supported by the Science and Engineering Research Council of United Kingdom. The author wishes to thank A.P. McCann and S. Fangohr for their comments and assistance in the preparation of this manuscript.

REFERENCES

2. K. Efe, "Heuristic Models of Task Assignment Scheduling in Distributed Systems", Computer, June 1982, pp. 50-56.
3. K. Efe, "User Friendly response Generation and Transparent Task Allocation in a Networked Abstract Machine Environment," PhD Thesis, Computer Science Dept. Leeds University, 1985.
4. V.B. Gyls and J.A. Edwards, "Optimal Partitioning of Workload for Distributed Systems," Proc. Compcon Fall 76, pp. 353-357.
5. K. Haessig and C.J. Jenny, "Partitioning and Allocating Computational Objects in Distributed Computing Systems," Proc. IFIP Congress 80, Melbourne, pp. 593-598.
6. S.P. Kartashev and S.I. Kartashev, "Distribution of Programs for a System with Dynamic Architecture," IEEE Trans. on Computers, Vol. C-31, No.6, June 1982, pp. 1-10.
7. M. Lan, "Characterization of Intermodule Communications and Heuristic Task Allocation for Distributed Processing Systems," PhD Thesis, UCLA, March 1985.
8. V.M. Lo, "Heuristic Algorithms for Task Assignment in Distributed Systems," Proc. 4th Intn'l conf. Distributed Computing Systems, San Francisco, May 14-18, 1984, pp. 14-18.
9. L.M. Ni, C. Xu, and T.B. Gendreau, "A Distributed Drafting Algorithm for Load Balancing," IEEE Trans. Software Engineering, Vol. SE-11, No.10, Oct. 1985, pp. 1153-1161.
10. J.A. Stankovic and I.S. Sidhu, "An Adaptive Bidding Algorithm for Processes, Clusters and Distributed Groups," Proc. 4th Intn'l. conf. Distributed Computing Systems, San Francisco, May 14-18, 1984, pp. 49-59.
11. H. Zimmermann, "OSI Reference Model-The ISO Model of Architecture for Open Systems Interconnection," IEEE Trans. Commun., Vol. COM-28, April 1980., pp. 425-432.

**Session 9: Reusable and Retargetable
Software**

**Chairperson: Phil Hwang
CNR**

Why Reusable Software Isn't

William J. Tracz

Program Analysis and Verification Group¹

Computer Systems Laboratory

Department of Electrical Engineering

Stanford University

Stanford, California 94305

(415) 497-1089

TRACZ@SU-SIERRA.ARPA

Abstract

Is reusable software a myth, magic or a messiah? This paper examines the issue of why the paradigm of reusable software engineering has not had the broad sweeping effects envisioned by the programming prophets. Parenthetically, the proverbial question "What makes reusing software artifacts difficult?" is raised and answered from the points of view of a programmer, software manager, computer scientist, and cognitive psychologist. Technical, organizational, political, and psychological bottlenecks are identified.

1. Introduction

There is nothing new under the sun. - Ecclesiastes 1:9

The concept of *reusable software* has been part of our programming heritage since the origins of the stored program computer EDSAC at the University of Cambridge in 1949. Maurice Wilkes¹ first recognized the need for avoiding the redundant effort in writing scientific subroutines and recommended that a library of routines be kept for general use. Until recently, little had been done to embellish this concept of program reusability over the 35 years that have passed. The formidable "software crisis" coupled with impressive improvements in the price to processing power ratio, advances in programming language design, compiler construction and interactive graphics has forced developers to reevaluate the tradeoffs made in establishing the traditional ad hoc development methodologies and environments used in the past. New and better ways are being explored to harness these recent technological advances and to develop an integrated software/hardware system optimized for programmer productivity. Again² and again³ the role of *Reusable Software* has been identified and discussed^{4,5,6,7,8}.

¹The author is an employee of the IBM Federal Systems Division, Owego, NY, participating in the IBM PhD Resident Study Program as a full time graduate student in the Electrical Engineering Department at Stanford University.

This paper examines the issue of why, to date, the paradigm of reusable software engineering has not had the broad sweeping effects envisioned by the programming prophets. Parenthetically, the proverbial question *"What makes reusing software artifacts difficult?"*, is raised, and answered from the perspective of a

- **Programmer,**
- **Software Manager,**
- **Computer Scientist, and**
- **Cognitive Psychologist.**

The goal of this paper is to identify the technical, organizational, political, and psychological impediments that have inhibited reusable software from being more prevalent in the state of the practice. Further discussion of these issues and their solutions, both implemented and proposed, may be found in a companion paper⁹.

The remaining portion of this paper is organized into two sections. The first contains a discussion of the various viewpoints regarding reusability. The second section contains the summary and conclusion.

2. Different Perspectives

If you are not part of the solution, then you may be part of the problem. - Anon.

"What makes reusing software artifacts difficult?" The answer to this question manifests itself in many technical, organizational, political and psychological issues. This section contains a discussion of the inhibitors and facilitators identified with reusable software presented from four points of view:

- **A Programmer:** someone who designs, implements and tests a portion of a software system.
- **Software Manager:** someone who manages a software development project.
- **Computer Scientist:** someone on the leading edge of technology, exploring and developing new techniques for expanding the reusable software engineering paradigm.
- **Cognitive Psychologist:** someone who understands the human thought process, its limitations and implications on programming.

2.1. A Programmer's Viewpoint

What are some of the reasons a programmer doesn't use someone else's code or design?

- It is more fun to write it oneself.
- The Not Invented Here syndrome of making oneself indispensable, or fear that in reusing someone else's code, it would be showing signs of weakness in not being able to do it oneself.
- It is easier to write it oneself, then to try and find it, figure out what it does, and if it

works. Furthermore, what are its attributes (performance) and dependencies (restrictions)? If it has to be modified, then it also might be easier to rewrite it.

- There are no tools to help find components, or compose a system from the reusable pieces.
- There are no known software development methodologies that stress reusing code, let alone reusing a design, or a specification?
- Little emphasis in reusing components is taught in academia¹⁰, in fact, most students don't have any mechanism or motivation to save programs from assignment to assignment, let alone, from course to course.

The technical issues raised here focus on the lack of well described, useful, and reliable reusable component libraries and an integrated programming environment available to take advantage of them. On a more philosophical note, the reluctance of a programmer to re-tool and place a dependence on someone else's work generally inhibits initial acceptance of this approach.

2.2. A Manager's Viewpoint

Managers often make decisions that are not based on technical issues alone. Some reasons for not adopting a Reusable Software Engineering approach for a software project might be as follows:

- If no tools or components exist, then it will take time and manpower to create the tools and the expertise. Such costs are generally not within the budget of a single project.
- If the tools truly do exist for making programmers more productive, then that will make the project more dependent on fewer personnel, consequently increasing the risk, and decreasing the number of people on the project (or reducing the empire a manager commands)⁸.
- Are the reusable tools deliverable items? Does the customer expect to need them to do maintenance?
- How does one set and maintain standards to control what is entered into the components library¹¹.

Technical issues faced by management are sometimes tainted by political considerations, or personal aspirations. Nevertheless, budgeting, scheduling and managing a software project based on a reusable components library requires a certain amount of confidence and experience in the methodology.

2.3. A Computer Scientist's Viewpoint

A computer scientist might take a more far reaching perspective when facing the issue of reusability. Balzer⁷ has stated that "Code is not reusable", suggesting instead of the *black box*, plug compatible approach focused on programming *products*, the answer to reusable software lies

in analyzing the programming *process*. From this perspective, the following alternative approaches to reusability have been suggested:⁵

- Very High Level Languages (VHLL's) that allow specification of problem domain entities and operations directly in the syntax of the language. Also, Problem Oriented Languages (POL's) are a form of VHLL's that are specifically tailored for a particular problem domain. Reusability is accomplished by reusing the compiler.
- Application generators are software tools that create programs given a parameterized or programmed specification. Reusability is again accomplished by reusing the application generator for each new problem.
- Transformation Systems require high level specifications be written describing *what* the software system should do. The specifications are then transformed by a series of pattern matching expansions into a program¹².

The key concept in each of these three examples focuses on the automated application of reusable components. Each tool recognizes some type of high level pattern in the problem domain which can be implemented by substituting some (parameterized) code fragments. Certain theoretical limits of the transformational techniques have been investigated. Another approach uses formal specifications for describing each component^{13,14}.

2.4. A Cognitive Psychologist's Viewpoint

Computer programming is one form of problem solving. Much insight has been gained in understanding the merits of several programming paradigms from the perspective of cognitive psychology¹⁵. Reusable software has been the focus of studies by Soloway¹⁶ and Curtis¹⁷. A summary of the empirical evidence gathered as it applies to reusable software engineering is as follows:

- The human thought process is limited, by the size of Short Term Memory, to the number of pieces of information it can manipulate consciously at one moment in time (7 ± 2)¹⁸. This complexity limit can be overcome by proper *chunking* or modularization of components, that is, collecting units of information together into one semantically meaningful piece (or package) (an argument that maps nicely into information hiding and object oriented design¹⁹).
- Expert (experienced) programmers develop applications through a recursive mental process²⁰ of matching pieces of the problem with solution segments which they are familiar with (plans²¹).
- Internal conceptualization of the knowledge base in which program/design segments reside tends to evolve with experience, into having a uniform content for all programmers²². In other words, experienced programmers tend to think alike, and express their solutions in similar forms.
- Programmers cannot reuse something they don't understand. Furthermore, expert programmers follow certain explicit *rules of discourse*¹⁶ regarding naming

conventions and programming style²³ which enhance program readability and comprehension.

These results support the need for a proper programming environment to facilitate the reusable software engineering paradigm. Tools must be available to handle the complexity, and assist the programmer in finding and understanding what software components exist. These results also demonstrate the intuitive validity of such an approach.

3. Conclusion

Those who cannot remember the past are condemned to repeat it - George Santayana

This paper has described the difficulties that arise when attempting to reuse software artifacts. The major issues may be summarized as:

- Most programmers tend to view reusability from the perspective of simply reusing code when reusing other programming artifacts (designs, specifications, and tests) leads to a more productive environment. Furthermore, other reusability paradigms (Application Generators, Translation systems, VHLLs, and POLs) have proven successful⁵.
- Useful, properly documented, tested, verified and classified reusable components need to be developed before they can be reused.^{24,7}
- Expert (experienced) programmers, with an understanding of the problem domain, and component library are best suited for fully exploiting the reusable software engineering paradigm.¹⁶
- Tools and methodologies need to be developed to support the development and cataloging of reusable components and the composition of software systems from them.¹¹
- There are staffing risks associated with a component based approach due to increased dependence on a single individual to do the work of many.⁸
- Reusable software development systems cost money, time, and manpower to develop and become proficient at using.

The results should in no way be interpreted as being insurmountable as various successful production quality systems^{25,26,27,28} and prototypes^{29,12,30,14}, have been implemented, or proposed. In particular work^{24,31} in Ada² is very promising. Further discussion of these systems is not within the scope of this paper and may be found in⁹.

²Ada is a registered trademark of the U.S. Government-Ada Joint Program Office

References

1. Office of Eames, editors, *A Computer Perspective*, Harvard Press, 1973.
2. Druffel, L.E. Redwine, S.T. Riddle, W.E., "The STARS Program: Overview and Rationale," *Computer*, Vol. 16, No. 11, November 1983, pp. 21-29.
3. Alexandridis, N.A., "Adaptable Software and Hardware: Problems and Solutions," *Computer*, Vol. 19, No. 2, February 1986, pp. 29-39.
4. Freeman, P., "Reusable Software Engineering: Concepts and Research Directions," *Proc. ITT Workshop on Reusability in Programming*, September 7-9 1983.
5. Horowitz, E. and Munson, J.B., "An Expansive View of Reusable Software," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, September 1984, pp. 477-487.
6. Jones, T.C., "Reusability in Programming: A Survey of the State of the Art," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, September 1984, pp. 488-493.
7. Standish, T.A., "An Essay on Software Reuse," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, September 1984, pp. 494-497.
8. Rauch-Hindin, W.B., "Reusable Software," *Electronic Design*, Vol. 31, No. 3, February, 3 1983, pp. 176-193.
9. Tracz, W. J., "Reusable Software Engineering: Issues and Answers", In progress
10. Denning, P.J., "Thowaway Programs," *Communications of the ACM*, Vol. 24, No. 2, February 1981, pp. 259-260.
11. Chandersekaran, C.S., and Perriens, M.P., "Towards an Assessment of Software Reusability," *Proc. ITT Workshop on Reusability in Programming*, September 7-9 1983.
12. Cheatham, T.E. Jr., "Reusability Through Program Transformations," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, September 1984, pp. 589-594.
13. Luckham, D.C., and von Henke, F. W., "ANNA: A Language for Annotating Ada Programs," *IEEE Computer Society Conference on Ada Applications and Environments*, October 15-18 1984.
11. Goguen, J.A., "Reusing and Interconnecting Software Components," *Computer*, Vol. 19, No. 2, February 1986, pp. 16-28.
15. Tracz, W.J., "Computer Programming and the Human Thought Process," *Software-Practice and Experience*, Vol. 9, 1979, pp. 127-137.
16. Soloway, E. and Ehrlich, K., "Empirical Studies of Programming Knowledge," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, September 1984, pp. 595-609.
17. Curtis, B., "Cognitive Issues in Reusability," *Proc. ITT Workshop on Reusability in Programming*, September 7-9 1983.
18. Miller, G.A., "The magical number seven plus or minus two: some limits on our capacity to process information," *Psychological Review*, Vol. 63, 1956, pp. 81-97, not yet received
19. Parnas, D.L., Clements, P.C., and Weiss, D.M., "Enhancing Reusability with Information

Hiding," *Proc. ITT Workshop on Reusability in Programming*, September 7-9 1983.

20. Jefferies, R., Turner, A.A, Polson, P.G., and Atwood, M.E., "The Processes Involved in Designing Software," in *Cognitive Skills and Their Acquisition*, Anderson, J.R., ed., Hillsdale, N.J.: Erlbaum, 1981, not yet received
21. Soloway, E. and Ehrlich, K., "What Do Programers Reuse? Theory and Experiment," *Proc. ITT Workshop on Reusability in Programming*, September 7-9 1983.
22. McKeihen, K.B., Reiman, J.S., Rueer, H.H., and Hirle, S.C., "Knowledge organization and skill differences in computer programmers," *Psychological Review*, Vol. 13, 1981, pp. 307-325, not yet received
23. Kernighan, B. and Plauger, P., *The Elements of Style*, New York: McGraw-Hill, 1978.
24. St. Dennis, R. Stachour, P., Frankowski, E., Onuegbe, E., "Measurable Characteristics of Reusable Ada Software," *Ada Letters*, Vol. 5, No. 2, March-April 1986, pp. 41-49.
25. Lanergan, R.G. and Grasso, C.A., "Software Engineering with Reusable Design and Code," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, September 1984, pp. 498-501.
26. Matsumoto, Y., "Some Experience in Promoting Reusable Software: Presentation in Higher Abstract Levels," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, September 1984, pp. 502-512.
27. Kernighan, G. W., "The Unix System and Software Reusability," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, September 1984, pp. 513-518.
28. Cavaliere, M.J., and Archambeault, P.J., "Reusable Code at The Hartford Insurance Group," *Proc. ITT Workshop on Reusability in Programming*, September 7-9 1983.
29. Neighbors, J.M., "The Draco Approach to Constructing Software from Reusable Components," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, September 1984, pp. 564-573.
30. Goguen, J.A., "Parameterized Programming," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, September 1984, pp. 528-543.
31. Mendal, G.O., "Designing For Ada Reuse: A Case Study," *Proc. of IEEE Computer Society Second International Conference on Ada Applications and Environments*, April 1986.

Towards Reusable Software Designs & Implementations

Ralph E. Johnson
Simon M. Kaplan
Department of Computer Science
University of Illinois at Urbana-Champaign
1304 West Springfield Avenue
Urbana, Il 61801

May 3, 1986

Abstract

One of the best ways to reduce the cost of producing software is to reuse old software instead of writing it anew. Although reusing small pieces of programs has long been practiced and software packages are also reused in their entirety, other products of the software design process, such as the general structure of a program or the design itself, are not formally reused. This paper discusses a new approach to the design and implementation of software systems. In this approach, a design is a formal mathematical object which can be instantiated as an object-oriented implementation. This allows reuse of both designs and implementations.

1 Introduction

The skyrocketing costs of software production are not helped by the fact that new software development often seems to require implementation "from the ground up" despite the fact that there are usually many parts of other systems that are similar to (but not the same as) portions of the new system under development. Also there seems to be a logical and philosophical divergence between the design of most systems and their implementations; designs are used to map out a system but frequently there is little correlation between the design and the finished product. Certainly there is no way to measure this correlation and no way to find out what decisions were made in going from the design to an implementation.

We need to gain a better understanding of the design phase, and also a better understanding of the implementation phase. We propose two orthogonal approaches to this: first, a formal model of design; and second an incremental approach to the implementation of designs, where a hierarchy that contains history information concerning the program development is encoded. The advantage of formally understanding design is that we reduce the chance of design errors and the chance of design

alterations that have severe impact on software. The advantage of hierarchy in the implementation process is that we can determine the correlation between design and implementation and also identify the minimum work required to reuse and retarget software.

The body of this paper consists of two sections. In the first we discuss our notion of formal support for software design and in the second we discuss our hierarchical model of software development. This model is closely akin to the class hierarchy of object-oriented systems with class inheritance.

2 Formal Support for Software Design

Engineering any system has two components; a *creative* component, and a *formal* component. The creative component is the intuition, insight and ideas that go into the design. The formal component is the checking of the design for robustness, completeness and soundness. These two components are inextricably intertwined; as the designer is working creatively, so he is bearing in mind the formal analyses which are to come, and trying to anticipate and prevent any flaws from entering the design which would be detected by the formal analysis. A better understanding of the formal aspects of software design, based on discrete mathematics, will allow a similar approach to the engineering of software systems – a creative process backed up by the ability to construct formal models of designs and reason about their correctness.

The major design activity is decomposing a problem (i.e. a specification for a system) into a set of smaller problems and interconnecting the solutions to these smaller problems to form a solution to the original problem. A designer recursively decomposes a high-level design into specifications for smaller and smaller components until the remaining components can be easily implemented or can be reused from some other system. Thus, a design describes the decomposition of a system.

Since the final implementation is described by a programming language, it is attractive to use the same language to describe the design, e.g. using Ada¹ as a design language. However, this artificially makes designs language-specific, uses a notation that is difficult to analyze formally, and uses the programming language for purposes for which it is usually ill-suited. Other, more intuitive, design techniques[11] lack a basis for formal rigor, as do most of the module interconnection languages[10]. We consider the components of a software system to be *algebras* (sets of functions on specific data types), and describe them with the language of discrete mathematics, in the style of [1].

Consider the system pictured in figure 1. In the algebraic view, the circles represent algebras. Each algebra supplies a specific part of the functionality of the total system, by supplying a set of operations on a specific set of data types. In Ada, these algebras would be implemented as packages. The operations would be procedures or functions and the data types will be Ada types or more complex types supplied by other packages. The Ada *package specification* corresponds to the definition of

¹ Ada is a trademark of the United States Government, Ada Joint Program Office.

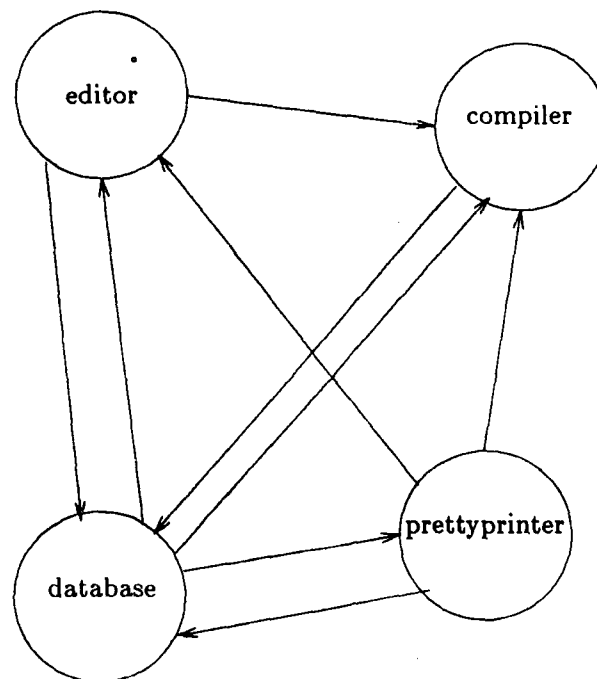


Figure 1: Schematic Representation of Component Interconnection

the signature of an algebra, and the *package body* corresponds to an implementation of the algebra. Ada does not provide the ability to describe the *semantics* of the algebra as part of the package specification. Languages such as Anna [7] would be employed to do this. The arrows between algebras represent *potential communications* between the algebras. For example, the compiler could be able to read source files from the database and write object files to it.

Having constructed this model of a design, we can ask questions of it, such as: Are all the software objects well defined? Is it possible for the components of the system that are supposed to communicate to do so? An algebra is well defined we can give a meaning to any sentence constructable from the signature [1]. The data values produced by one algebra can be used by another if there is a function that maps elements from selected carriers of the first algebra to carriers of the second. Thus, we can answer a number of questions about designs by translating them to questions about the equivalent algebras.

There are some questions about designs that can be answered by considering the general model of design representation that we have constructed without any reference to specific designs. For example, most software development environments use some universal language for communication between tools in the environment, such as DIANA. Investigation into the formal structure underlying our design model has suggested that universal languages should not be used for this purpose as they needlessly limit the flexibility and reusability of tools which rely on them. (DIANA,

for example, used an attribute-grammar based paradigm. Attribute-grammars have not yet been shown to be efficient enough for future software development tools). Instead of a universal language there should be a (small) number of communications mechanisms which allow communication between tools and support conversion between the types used by the various tools in the environment.

Because the design objects are algebras, they are often themselves executable[4]. This means that we can test designs by executing them. We can also test their correct interaction by using systems that test module interconnection such as Polyolith [8]. This is obviously advantageous to the software engineer, since the costs of rectifying design errors once a system has been constructed are high.

Once we are satisfied with a design, an implementation can be constructed. An approach based on *category theory*[6][5] allows us to reason formally about the instantiation of an implementation from a design. The design is viewed as a blueprint for the implementation. Coupling the design and implementation closely in this way has several advantages from the software engineering viewpoint. First, some aspects of the implementation process can be automated, such as automatic choice of control structures by analysis of the specifications. Second we can record implementation decisions as they are made when instantiating the implementation from the design (such as choice of data representations).

We believe that "software reuse" and "design reuse" are inherently inextricable. In our model of program development, design and implementation are very closely bound. The fundamental focus for reuse should be *design* reuse. Each component of a design has an implementation (or possibly several implementations) closely bound to it, together with a history of how those implementations were derived from the design. Each component may also have a design, which will be part of the derivation history of the component. By reusing design components, one automatically gets the implementations. These implementations can be reused if the design is unaltered, or rapidly modified if the design is changed.

3 Object-Oriented Realizations of a Formal Design

An *object-oriented* implementation is natural for our design blueprints. The circles and arrows in the design are objects in the implementation, where the objects which represent the circles provide the required functionality of the system and the objects which represent the arrows are messages between the functional objects.

One method of object-oriented programming is that used in Ada. Here, packages are used to encapsulate objects, providing information hiding, abstraction and the functionality of an object in a design. Because communication between objects is limited to procedures described in interface specifications, clean interfaces between program components are more likely to arise than if the program were designed as a set of subroutines. Much of the design of a system is expressed in the interface specifications of its packages, since the interface specifications indicate which package implements each function that the system is required to implement. This is why designs and implementations can be tightly bound; the package specification

describes the signature of the design object (*i.e.* algebra) which it implements, while the package body describes the algebra itself by means of a concrete implementation. This style of programming is supported not only by object-oriented programming languages, but by many other modern programming languages.

Many object-oriented programming languages provide polymorphic functions (provided to a limited extent in Ada by the overloading and generic facilities of that language) which greatly increase the potential for code reuse. Because objects in these languages interact only by sending messages to each other, objects depend only on each other's signatures, *i.e.* the set of operations that they have defined. The implementation of the operations for any object is defined by that object's *class*. For example, a printer object can print any object whose class has defined an operation which returns a printable representation of the object.

Polymorphism is most useful when there are families of classes with similar signatures. Programmers tend to implement those operations necessary to let new classes be used with existing polymorphic functions, resulting in these families of classes. This not only makes implementation easier but makes the resulting code easier to reuse. Thus, polymorphism increases the reuse of both code and design.

Many object-oriented languages encourage the creation of families of classes that support similar operations by allowing one class to inherit the operations and data-structures defined by another. Families of classes form a hierarchy, each with a class that defines the operations common to the family. Often the top-most class defines operations so abstractly that they are unreasonably inefficient or even not executable, but the subclasses that make up the family redefine operations and pick specialized representations for data in order to reach an acceptable level of efficiency. The class inheritance hierarchy records the choices in data representation that were made and the algorithmic improvements that resulted. These are important design decisions that are usually never explicitly recorded in most other programming styles. This is vital for software reuse. If one takes a design and associated implementation, and makes a change to the design, then changes to key points in the inheritance hierarchy can retarget the implementation for the new design.

The top-most class of a family sometimes acts as a code template, further encouraging code reuse. It may leave a few crucial operations undefined, but define a great many more operations in terms of the few undefined ones. Subclasses will define the missing operations and so be able to use the many operations defined by the top-most class. Thus, by defining a few operations for a new class, a programmer can make use of many other operations. If a new operation is defined for the top-most class then its subclasses immediately inherit that operation. This contrasts with the ad-hoc way in which code templates are usually implemented.

Another way in which code templates are useful is to perform the refinements of an object (the creation of lower levels in the hierarchy) using an *abstract programming language*. This means that a development can be retargeted to different programming languages by instantiating the abstract program into different concrete programming languages (such as Ada, Modula-2 or CLU).

Object-oriented programming languages have a reputation for encouraging code

reuse[9]. This is in part because they encourage data abstraction, partly because the class hierarchy allows designs to be recorded, and partly because of the programming environments of which they are usually a part. However, these languages are usually designed for the unconstrained style of artificial intelligence programming rather than the more rigid demands of software engineering.

Some work has been done on improving the object-oriented languages[3], but it is also possible to add class hierarchies to languages designed for software engineering. A database that stores multiple versions of a program can be used to let a language like Ada mimic the class hierarchy of an object-oriented language. The superclass-subclass relationship can be represented by the old version-new version relationship. A subclass inherits all the operations of its superclass, but adds new operations and changes some of the old operations. In the same way, the new version of a package can add new operations and change old operations. This technique is likely to use multiple versions of a package at the same time, but few version control systems permit several versions of an object to be used simultaneously. This suggests that a specialized version control system might be needed to fully support class hierarchies in languages of which they are not normally a part.

4 Future Research

Most research into algebraic abstract data-types has been primarily theoretical, with a few notable exceptions[4][2]. We are investigating how that theory can be used to solve the important design problems in software engineering, such as those discussed above. By gaining an insight into the formal relationships between components of a design we get a better idea of just how to perform the decomposition process. By understanding design decomposition we understand better how to specify and use modularity, for improved program development productivity, reliability and reusability and retargetability. All of these are intricately coupled issues.

Most work with object-oriented programming has been by programming language designers or by those interested in applying object-oriented programming to particular areas such as artificial intelligence, office automation, or computer aided design. We are investigating object-oriented programming from the software engineering viewpoint and expect to use it to record design decisions and enhance code reuse.

In particular, we are investigating the following problems:

1. Which design problems can be solved using techniques borrowed from discrete math?
2. Which of the many methods for describing algebras form the best basis for specification of programs?
3. How does our design technique influence the design of a module interconnection language.

4. How can our design technique be used with modern programming languages developed for software engineering?
5. How can we best use the ability of a class hierarchy to record design decisions and provide code templates?
6. What are design principles for class hierarchies that maximize code reuse?
7. How should programming environments be changed to support and take advantage of object-oriented programming?

We believe that our hybrid approach to these problems – combining research into formal and informal issues – will allow us to gain insight into solutions to pressing software design and implementation issues and apply this insight to the development of practical solutions to these problems.

References

- [1] J. A. Goguen and R. M. Burstall.
Introducing institutions.
In E. M. Clarke and D. Kozen, editors, *Logics of Programs, Lecture Notes in Computer Science 164*, Springer-Verlag, New York, 1984.
- [2] J. V. Guttag, J. J. Horning, and J. M. Wing.
Larch in Five Easy Pieces.
Technical Report 5, Digital Equipment Corporation Systems Research Center, July 1985.
- [3] R. E. Johnson.
Type-checking smalltalk.
In *To appear in proceedings of Conference on Object-Oriented Programming Systems, Languages and Applications*, 1986.
- [4] S. N. Kamin, S. Jefferson, and M. Archer.
The role of executable specifications: the fase system.
In *Proceedings of the IEEE Symposium on Application and Assessment of Automated Tools for Software Development*, November 1983.
- [5] S. M. Kaplan.
The isep open systems architecture: a categorical perspective.
In Preparation.
- [6] S. M. Kaplan, R. H. Campbell, M. T. Harandi, R. E. Johnson, S. N. Kamin, J. W-S. Liu, and J. M. Purtilo.
An architecture for tool integration.
In *Proceedings of the International Workshop on Advanced Programming Environments*, Trondheim, Norway, June 1986.
To Appear as Springer-Verlag Lecture Note in Computer Science

NO-A184 949

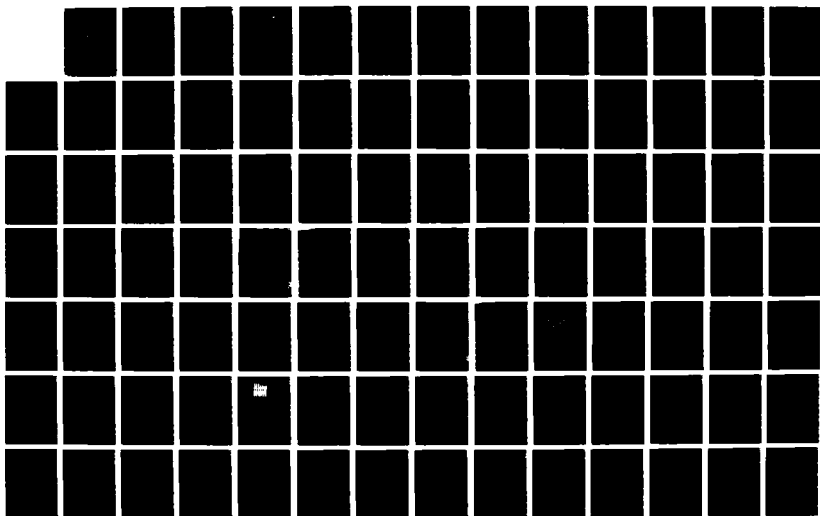
PROCEEDINGS OF THE WORKSHOP ON FUTURE DIRECTIONS IN
COMPUTER ARCHITECTURE. (U) BATTELLE COLUMBUS LABS
RESEARCH TRIANGLE PARK NC D P AGRAWAL ET AL. 30 AUG 86
ARO-86384-EL DAAG29-81-D-0100

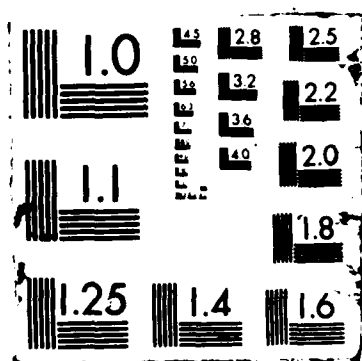
3/5

UNCLASSIFIED

F/G 12/5

NL





- [7] D. Luckham and F. von Henke.
An overview of anna, a specification language for ada.
IEEE Software, 2(2), March 1985.
- [8] J. M. Purtilo.
Polyolith: an environment to support management of tool interfaces.
In *Proceedings of the ACM SIGPLAN Symposium on Language Issues in Programming Environments*, Seattle, WA, December 1985.
- [9] M. Stefik and D. G. Bobrow.
Object-oriented programming: themes and variations.
The A.I. Magazine, VI(3):40-62, Winter 1986.
- [10] W. F. Tichy.
Software Development Control Based on System Structure Description.
Technical Report CMU-CS-80-120, Department of Computer Science, Carnegie-Mellon University, 1980.
- [11] E. Yourdon and L. L. Constantine.
Structured Design.
Prentice-Hall, Englewood Cliffs, 1979.

Archotyping - A Knowledge-Based Reuse Paradigm

Stanley M. Przybylinski
General Dynamics Data Systems Division
P. O. Box 85808
V2 - 5530
San Diego, California 92138

ABSTRACT

The "software crisis" has made firms who depend on hardware and software systems for their livelihood (read "everybody") keenly aware of the need to improve programmer productivity and successfully capture the knowledge gained on previous software development projects. Several government and industry initiatives (Software Technology for Adaptable, Reliable Systems (STARS) and the Software Productivity Consortium (SPC), respectively) have focused on software reusability as one solution to this problem.

DARTS™, a proprietary software engineering environment developed by General Dynamics' Data Systems Division, provides a unique answer to the reuse question: archotyping, the use of "embedded" high-order language statements and an advanced macro-processing capability to automatically generate new versions of complex software systems. This paper will provide an overview of the technology and archotyping, including some examples of archotyped code. It will also discuss other knowledge-based applications of DARTS™.

BACKGROUND

The Development Arts for Real-Time Systems (DARTS™) Technology began development in mid-1979 as a General Dynamics Data Systems Division software initiative to try to solve the software/programmer productivity problem. DARTS™ is a knowledge-based software engineering technology conceived and developed to implement a Software Manufacturing System (SMS). This system allows domain experts (e.g., missile designers, C³I system designers, etc.) to generate new versions of complex software systems without programmer intervention using specification languages.

The implementation of this software manufacturing concept required a new, very high-order language incorporating many high-technology software approaches (e.g., pattern matching, dynamic knowledge representation, list processing, set processing, embedded languages, dynamic code construction, compilation and execution) into one self-defining, extensible language. The embodiment of these needs became the Archetype Xenoclause Embedding (AXE™) Language.

THE CONCEPT OF "ARCHETYPING"

The basic premise of the SMS, shown in Figure 1, is to capture existing, fully tested and delivered software soon after its completion. The same set of software engineers who developed the original system then "archetype" it. Archetype comes from the Latin *archetypum* for "first molded as a pattern, exemplary." In this case it refers to a process where the software specialists work along with users of the system (and potential users of similar systems) to

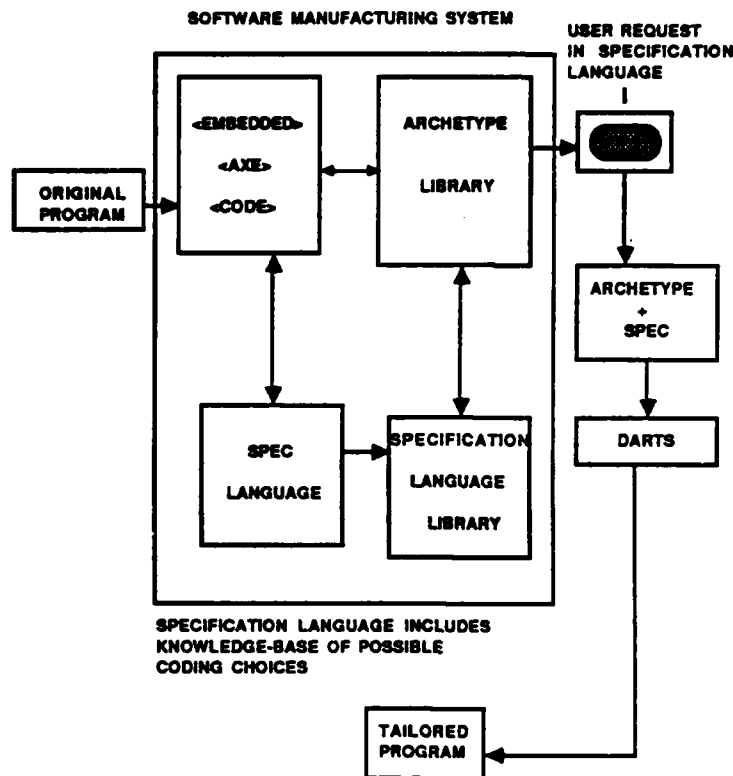


FIGURE 1 - DARTS IMPROVES PRODUCTIVITY THROUGH CAPTURE OF DOMAIN AND PROGRAMMER KNOWLEDGE

sketch out possible future requirements for systems of this type. Thus, this team of experts develops a "pattern" from which future systems can be generated, hence the use of the term "archetype."

Archotyping is achieved through the use of "xenoclauses" (xeno- , from the Greek *xenos* for "stranger"), foreign clauses (or statements) "embedded" within the native code which represent areas of possible change. Xenoclauses can be embedded in any native language. The code sections that replace the xenoclauses to effect these changes are stored in a knowledge-base, a very flexible storage system that is an inherent part of the DARTS™ Technology. This explains the reason for the acronym "AXE™."

The last step is identifying these changes and implementing them without programmer intervention. Software engineers, working with domain experts, develop a "specification language," a set of terms and parameters which have special meaning within a particular problem domain. This implies that a different specification language must be built for each domain. This is different from the normal software engineering concept of a single specification language for all domains. The replacement code sections are stored in the system knowledge-bases under access criteria referenced by the specification language.

As new systems are required, users specify their requirements, which causes archetypes (and their corresponding knowledge-bases) to be accessed from the SMS libraries. These library entries and the user's specification form the inputs to DARTS™. The archetyped files are interpreted by the BOLT™ Processor, the interpreter of DARTS™, with the embedded AXE™ Language statements replaced by code sections from the knowledge bases.

The end result of this process is a fully functional program ready for compilation. This newly generated code meets all stated specifications. Since this new version is based, for the most part, on a fully tested original, debugging time is minimized. If a coding error recurs because an archetyped code section is in error, it can be repaired by changing the knowledge base and regenerating the program.

To understand how the technology works, it is necessary to provide a brief outline of AXE™ language syntax and usage.

AXE™ LANGUAGE CONCEPTS

AXE™ language statements are demarcated by a less-than ("**<**") character on the left and a greater-than ("**>**") symbol on the right:

< {AXE™ statement} >

All AXE™ statement processing either replaces a statement with some symbol or leaves no symbol behind (replaces a statement with a "null"). The BOLT™ Processor also ignores any characters in the input file that are outside the boundaries of an AXE™ statement and passes them unchanged to the transformed input. For example, suppose the **<{AXE™ statement for a greeting}>** represented the word "HELLO" and a **<{Statement giving a person}>** represented "READER." Then the BOLT™ Processor would convert:

SAY <{AXE™statement for a greeting}>TO THE<{Statement giving a person}>

into:

SAY HELLO TO THE READER

AXE™ statements can be continued across multiple lines using the "**<\$>**" and "**<\$\$>**" statements. "**<\$>**" means "ignore all characters to the right of me." "**<\$\$>**" means "ignore all characters to the left of me." These are useful for enhancing readability and inserting comments and other information into AXE™ code. For example, the sample statement used above could also be written:

**SAY <{AXE™ statement for a greeting}> TO<\$>
<\$\$> THE<{Statement giving a person}>**

SYMBOLIC REFERENCING

AXE™ is an object-oriented language, i.e., all pieces of information can be accessed as "metasymbols," symbolic references to data, lists, sets, procedures or just about anything. Unlike variables or symbols in other programming languages, metasymbols can have more than one value, depending on the value of user-specified access conditions. These conditions, called "CONTEXT" and "CRITERIA" expressions, must be satisfied for a particular metasymbol value to

be accessed. The use of these expressions is similar to the concept of frames in Artificial Intelligence (AI).

As a simple example, suppose that a FORTRAN program called for the computation of "distance equals 1/2 times gravity times the square of time":

$$1000 \text{ DIST} = 0.5 * \text{GRAVITY} * \text{TIME}^2$$

This statement can be generalized using DARTS™. Assuming the contexts "EARTH" and "MOON" have already been created, the DEFINE blocks:

```
<DEFINE :GRAVITY> <IN-CONTEXT ":EARTH">  
<START-VALUE>32.2<END-VALUE>  
<END-DEFINE>
```

```
<DEFINE :GRAVITY> <IN-CONTEXT ":MOON">  
<START-VALUE>5.6<END-VALUE>  
<END-DEFINE>
```

create a knowledge-base of gravity values. The FORTRAN statement is now written as:

$$1000 \text{ DIST} = 0.5 * \{ \text{AXE}^{\text{TM}} \text{ syntax for "Replace"} \} \text{GRAVITY} * \text{TIME}^2$$

(Note: Since many languages include "<" and ">" in their character set, translation map routines have been built using AXE™. Source code must be preprocessed using these routines to remove the "<"s and ">"s and postprocessed to return them where required.)

Using an AXE™ statement or a specification language, the context "EARTH" is established. The BOLT™ Processor would then interpret the above statement and result in the FORTRAN line:

$$1000 \text{ DIST} = 0.5 * 32.2 * \text{TIME}^2$$

With this technique software engineers can code to specification, with variations on the theme implemented using the knowledge-base.

This simple example shows only a small sample of the power of this technique. The metasymbol state may be a complex boolean expression of CRITERIA, with an upper limit of 4096 different conditions. The "value" between <START-VALUE> and <END-VALUE> can be an AXE™ statement itself, resolved into a numerical constant prior to substitution into the FORTRAN expression. Thus, in the previous example, TIME could be changed to an AXE™ statement representing the value obtained by a call to a real-time clock or some other routine.

Metasymbols can also be entire code sections, replacing symbol references according to the indicated context. This capability was used during a recent rehost of DARTS™ from the DEC VAX to an IBM 3081. The sections of DARTS™ that were VAX architecture-dependent were identified and archetyped, with replacement code sections written that were targeted for the IBM architecture. This method can be very beneficial when many slightly different versions of substantially "the same" software must be maintained.

LIST EXPANSION

List-driven expansion is another powerful AXE™ technique useful for generating multiple

copies of similar character strings. AXE™ lists have the following syntax:

`<LIST {Delimiter definition} COMPONENT1 {delimiter}...>`

where {Delimiter definition} is the first non-blank character after LIST. Most special characters can be used as delimiters. Individual LIST components can be data structures of any kind. For example:

`<LIST/1/2/This is a LIST/3/>`

is a legitimate AXE™ list.

In list-driven expansion, special AXE™ functions define a "model" copy of some AXE™ program fragment. The model tells AXE™ how many copies of itself to repeat. Each new copy is generated by inserting different text at predetermined points within the model and concatenating the resulting new copy to the previous one. The final concatenated block of generated statements replaces the model itself in the program in question.

A "model" is any section of an AXE™ program headed by a special function called the "expansion" function ("`<[>`"). The model terminates with a "model termination" function ("`<]>`"). Actual code might look like:

`<[> I am a base model used for the expansion.<]>`

Each part of the model that changes from copy to copy is built from within the model by an "inner expansion" function, which is an "expansion" function ("`<[>`") supplying a list of strings:

`<{AXE™ execution information}[[strings]]>`

Continuing with the model above, an expansion list could be:

`<[,first,second,third,]>`

The word LIST is not necessary here because the use of "`<[`" implies that a list follows. In this case a "," delimits the LIST components. An "inner expansion" function placed within a model directs AXE™ to generate multiple copies of the model. For example, the statement:

`<[> I am a base model used for the <[,first,second,third,]> expansion. <]>`

would result in

I am a base model used for the first expansion. I am a base model used for the second expansion. I am a base model used for the third expansion.

As an example of this, consider the following skeletal Pascal program with embedded AXE™ statements:

```
i
PROGRAM COLORDemo(INPUT,OUTPUT)
```

```

CONST NUMCOLORS = 3;
:
TYPE  NAME = CHAR;
      COLORTYPE = ARRAY[1..NUMCOLORS] OF NAME;
:
VAR COLORNAM : COLORTYPE;
:
PROCEDURE INIT;
VAR I,J : INTEGER;
BEGIN
<[>  COLORNAM[<[1...,]>] := '<EXPAND \:COLORLIST>';<EOL>
<]>
:
END
:

```

with the following information stored in the knowledge-base:

```

<DEFINE :COLORLIST>
<IN-CONTEXT ":PRIMARY">
<START-VALUE><LIST|RED|YELLOW|BLUE|><EXIT-VALUE>
<END-DEFINE>

```

The ellipsis ("...") is a default in AXE™ that tells the BOLT™ Processor to expand the list, starting at "1" in this case, to the length provided by the next list, COLORLIST here. (Note that the "|" was selected as the list delimiter for this example.) <EXPAND \:COLORLIST> means just that: expand COLORLIST and replace this expression as many times as is indicated by the list length. In this case 3 copies are produced. Thus, with the context set to "PRIMARY", processing this list expansion results in the following Pascal code:

```

:
COLORNAM[1] := 'RED';
COLORNAM[2] := 'YELLOW';
COLORNAM[3] := 'BLUE';
:

```

List expansion can be combined with symbolic referencing to further generalize the archetype. Instead of simply declaring its value as a constant at the start of the program, NUMCOLORS can be determined by obtaining the dimension (or number of elements) of COLORLIST in the proper context, i.e.,

```
CONST NUMCOLORS = <_DIMENSION.\:COLORLIST>;
```

DIMENSION is just one of the many useful metasymbol attributes readily available to AXE™ programmers.

Other values of COLORLIST can be added to the knowledge-base under different contexts, e.g.,

```
<DEFINE :COLORLIST>  
<IN-CONTEXT ":SECONDARY">  
<START-VALUE><LIST|ORANGE|GREEN|PURPLE|><$>  
<$$><EXIT-VALUE>  
<END-DEFINE>
```

```
<DEFINE :COLORLIST>  
<IN-CONTEXT ":ALL">  
<START-VALUE><LIST|BLACK|><EXIT-VALUE>  
<END-DEFINE>
```

```
<DEFINE :COLORLIST>  
<IN-CONTEXT ":NONE">  
<START-VALUE><LIST|WHITE|><EXIT-VALUE>  
<END-DEFINE>
```

Now, expansion under the context "PRIMARY" would result in the same Pascal code as above. If the context was changed to "SECONDARY", however, the expanded code would be:

```
;  
COLORNAM[1] := 'ORANGE';  
COLORNAM[2] := 'GREEN';  
COLORNAM[3] := 'PURPLE';  
;
```

Finally, setting the contexts "ALL" or "NONE" would result in:

COLORNAM[1] := 'BLACK'; and COLORNAM[1] := 'WHITE';

respectively.

As stated previously, these techniques can be applied to any source language. AXETM statements can be nested, with multiple CONTEXT's and CRITERIA's, resulting in an almost limitless number of archotyping combinations.

OTHER DARTS™ APPLICATIONS

The DARTS™ Technology has also been successfully used in a number of other application areas:

- Language Translators - the pattern-matching and knowledge-base capabilities of DARTS™, originally included to allow the parsing and comprehension of specification languages, make DARTS™ the ideal vehicle for language translation. This has been the most successful application area to date. Translators providing over 80% conversion rates have been brought up in periods ranging from 2 man-weeks to 2 man-months, depending on the languages involved. DARTS™ engineers work closely with customers to determine the most cost-effective translation

percentage, identifying the level and nature of the manual effort required to complete the conversion. Documentation, at least in a rudimentary form, can also be produced as a by-product of the translation process.

The list of completed translators includes CDC Assembly Language to Cray Assembly Language, Franz Lisp to Common Lisp, COBOL to FORTRAN, and Pascal to C. DARTS™ has been particularly useful in converting fourth generation languages good for prototyping (e.g., Mark IV, Model 204 User Language) into production quality systems (COBOL for these examples).

- Knowledge-Based System Development - The list/set processing and knowledge-base capabilities of DARTS™ also make it a useful tool for expert system development. However, DARTS™ is not an expert systems shell, like ART or KEE. Building an expert system using DARTS™ is just like building one from scratch using Lisp. Both require the same types of resources and development efforts. Knowledge engineers are a vital commodity. An inference engine must be constructed. The user interface must be defined and built.

On the other hand, DARTS™ does provide some advantages. Knowledge-bases and knowledge representation methods are built into the technology. While custom inference engines are necessary for each DARTS™-based expert system, experience on Lisp-based systems has shown that inferencing is most effective when tied directly to the knowledge-base structure. DARTS™ also provides a screen definition and processing capability that greatly simplifies user interface development.

A number of expert systems have been completed for General Dynamics internal projects, including a Design Analysis Expert System. This system was designed to analyze engineering drawing notes for completeness and internal consistency with required company and government specifications. The menu-driven system highlights errors and provides specification numbers and suggested corrections for each entry.

PRODUCT INFORMATION

DARTS™ is currently available under license from General Dynamics. DARTS™ includes the BOLT™ Processor, the AXE™ Language (AXEL) knowledge-base, and system documentation. The BOLT™ Processor consists of approximately 50,000 lines of PASCAL, FORTRAN and assembly language code and executes in a 8 MB virtual image on Digital Equipment Corporation VAX series computers under VMS. The IBM version, written all in Pascal, runs on the 308X series under MVS/XA. BOLT™ Jr., an OEM BOLT™ Processor, is an execution-only version to support secondary sales of programs/systems written in the AXE™ language.

SUMMARY

DARTS™ provides a knowledge-based solution to software reuse. The examples and AXE™ language information within were intended to provide insight into the power of archotyping, without getting bogged down in technical details. If this message came through, it is hoped that DARTS™, or at least these concepts, can form the basis for an innovative solution to the "software crisis."

**Session 10: Distributed Operating
Systems**

**Chairperson: Douglas Jensen
Carnegie-Mellon University**

Distributed Control of Large Parallel Computers

Larry D. Wittie

Associate Professor of Computer Science
State University of New York at Stony Brook
Stony Brook New York 11794-4400
(516) 246-8215/7146

Over the last three years there has been a resurgence of interest in the design of large parallel computer systems. Several new classes of highly parallel computers have recently been introduced. These include:

- (1) massively parallel hypercubes, such as the 64,000 1-bit computer SIMD network Connection Machine¹;
- (2) large multiprocessors with a few hundred processors sharing a global memory via a $\log(N)$ -stage interconnection network, such as the IBM RP3² and University of Illinois Cedar³, each with about 512 processors;
- (3) large multiprocessor/multicomputers with clusters of a few hundred computers sharing a common memory and linked to other clusters, such as Cedar³ and the planned Vitesse VNP^{4, 5} tree network of up to 1089 computers, linked within each cluster by a crosspoint switch to shared memory and between clusters as a tree of fanout 8 to 16.
- (4) small multiprocessors with a few dozen processors sharing memory via a fast global bus, such as the Encore Multimax⁶ and Alliant FX/8⁷;
- (5) large multicomputers of a few hundred computing nodes linked by hypercube interconnections, such as Ncube and the Intel/Caltech Cosmic Cube⁸;
- (6) massive SIMD array processors, such as the 11 Gflop IBM GF11⁹, built specifically for physics quark modelling; and
- (7) one to sixteen processor, shared memory versions of the ultrafast vector processors that are the current commercial supercomputers, such as Cray X/MP-4¹⁰, Cray2¹¹, Fujitsu¹², Hitachi¹², NEC⁵, CDC Cyber-Plus⁵, and the planned ETA-10¹³.

1. Operating Systems for New Parallel Machines

The operating systems of all these machines differ significantly. SIMD machines have only one instruction stream common to all processors. Resource management is a significant part of the algorithm programmer's task. Each algorithm must deal directly with allocation of data to memories and the movement of data to the processors when needed. The operating system for an SIMD machine usually runs on the general purpose host to which it is attached. It aids development of new algorithms, loads new jobs, and accepts the torrent of data produced during execution.

The new shared memory multiprocessor systems are a significant interim step in solving the programming problems posed by massively parallel multicomputers with separate memories for each processor. By having special hardware support for access to globally shared memory, they allow concurrent programs to have many processors passing common data rapidly via the shared memory. Programmers need worry only about how to subdivide their algorithms into separate processes. They need not worry about delays in exchanging data among processes. Communication delays are a major additional concern for distributed programs running on multicomputers without shared memory.

Multiprocessor programmers still must worry about decomposition of problems into processes, synchronization between processes, and partial failures by individual processors. Multiprocessors are only an interim step because rapid, frequent, simultaneous access to a globally shared random access memory by millions of processors is almost certainly an impossible task. With more than ten processors sharing memory, there are severe problems with memory bandwidth. More insidiously, when there are many processors, most may be waiting on locks to access shared data that another processor is changing.

The small multiprocessors (Encore, Alliant) use high-speed global busses and hardware support of synchronization and locking operations such as test-and-set for shared memory access by up to 20 processors. Further expansion of these systems will be very difficult, especially because of locking delays. The large multiprocessors (RP3, Cedar, Vitesse) all have permanent memory and recent access caches local to each processor to greatly reduce reading of shared memory. In many ways, they are multicomputers. They also have efficient interconnection networks to allow all processors to access different parts of shared memory simultaneously and very special hardware to minimize blocking during writes to common memory locations.

In particular, both the IBM RP3 and the Cedar multiprocessors use $\log(N)$ -stage Omega¹⁴ networks for N simultaneous processor-to-memory accesses. As justified by simulations for the NYU Ultracomputer¹⁵, several million dollars of the costs (\$20M) of the 512 processor, 800 Mflop RP3 is special network logic that combines simultaneous fetch-and-add (F&A) operations to the same memory location. The combining logic returns a consistent set of fetched values to the various processors and adds the sum of their increments to the memory location. The effect is exactly the same as if the simultaneous operations executed one after each other in some unspecified order. In particular, this F&A operation can be used to manage queues for operating systems such as Unix so that many processors can simultaneously add and delete entries without locks.

Almost all of the shared memory multiprocessors, including the Cray2 and Alliant vector processing systems, support the popular Unix operating system because it offers a flexible interface to users. With shared memory, it is easy to support parallel execution at the coarse-grained Unix process level. If there are enough processes, all processor can work simultaneously, except during synchronization waits. The vector processors also offer tools for extracting fine-grained concurrency from manipulations of matrices and vectors. For example, different Cray processors may be simultaneously executing the same Fortran DO-loop for different ranges of index values.

2. Operating Systems for Future Massively Parallel Machines

The current hypercube multicomputers (Cosmic Cube and Ncube) and the planned MIMD version of the Connection Machine are precursors of future massively parallel multicomputers ("megacomputers"). Frequent access to globally shared memory will be impossible in megacomputers. In the near-future, designs like the Vitesse system, which is a multicomputer tree of multiprocessors, are very likely to be the most efficient way to build machines with many thousands of processors. Clusters of processors can access a hierarchy of shared memories, with the most distant, globally shared memories accessed very infrequently.

There appear to be at least three avenues of study for algorithms to manage resources and to perform application calculations on megacomputers, where system size will lead to delays and other inaccuracies in data from distant computing elements:

- (1) distributed algorithms, in which separate processors use mainly local data augmented by infrequent accesses ("messages") to data in memories of distant processors;
- (2) probabilistic algorithms¹⁶, in which good approximations to exact results can be calculated rapidly from volumes of input data, even if a few items are incorrect.
- (3) asynchronous algorithms, in which correct results do not depend on locked step operations by different processors, avoiding wasteful waits to achieve synchrony.

Work at SUNY/Stony Brook on control algorithms for megacomputers has already started exploring two of these avenues, tree-based distributed control algorithms¹⁷⁻¹⁹ for networks of computers and asynchronous communication and data replication mechanisms²⁰⁻²³ for networks. Even computers built from processors clustered around locally shared memories have data sharing restrictions consistent with a model of (multiprocessor) computers embedded in a communication network.

2.1.1. Dynamic Group Organization and Use for Distributed Systems

An extensive study has been made of ways to implement and use multicast communication within groups of components in large netcomputers, especially ones linked by grids of ethernet. Techniques for organizing dynamic groups have been studied for use in supporting multicast communication, in maintaining consistent replicated data, and in providing network services. Early results have been published in three papers²⁰⁻²² and a dissertation²³.

2.1.2. Tree-based Distributed Control Algorithms

Work has begun in exploring the efficiency of several newly created distributed algorithms for resource management in large networks. Most of these algorithms use hierarchical patterns of communication to allow efficient centralized control of high-level performance. However, they are being analyzed both in terms of tree-branch distances and in terms of physical distances for the common cases where they are imbedded in regular physical topologies such as 2- or 3-dimensional nearest-neighbor meshes.

Because most of their data paths traverse an $O(\log N)$ number of tree branches, tree-based control algorithms seem suitable to a large number of different underlying network topologies. In other words, they hold promise of being nearly universally applicable

to a large class of network computers. Of course, raw state data from the lower levels of the tree must be compressed or filtered at each higher management level to avoid overloading nodes near the root.

The one completed study¹⁹ evaluates an algorithm for very locally reconfiguring a spanning tree of processors assigned network management duties after a failure has incapacitated one of them. The analysis techniques are able to give closed form expressions for the expected values of several measures of the algorithm, including:

- (1) the maximum extent of network disruption during the recovery after one failure,
- (2) the increase in total spanning tree branch lengths per local reconfiguration,
- (3) total message traffic generated to reconfigure after one failure, and
- (4) the total delay until the control tree is completely recovered from each failure.

There are a number of other distributed control algorithms for which some initial results are known. Each is based upon spanning trees of control processors. Each drastically restricts the flow of management data up the tree to avoid traffic bottlenecks near the root. They include:

- (1) two other algorithms (**Promotion** and **Chaining**) for local repair of spanning trees of processors used to control massive networks,
- (2) algorithms for finding near-optimal routes in large networks without using either global models of the topology or global broadcasts to find the shortest path from a source to a new destination,
- (3) algorithms for managing limited-size distributed caches mapping names to network identifiers for recently accessed resources so that requests for specific resources can usually be satisfied without expensive global searches,
- (4) algorithms for finding a nearby instance of a large pool of identical resources, such as an idle processor that can immediately execute a single task, and
- (5) an algorithm for finding a physically close group of idle processors for co-scheduling of several tasks that must be executed simultaneously since they interact closely.

2.1.3. A Comparison of Two Failure Recovery Algorithms for Trees

As a sample of the results that can be obtained from these analyses of specific algorithms, here is a summary of the characteristics of two algorithms. Each locally replaces a failed member of a tree of computers. The computers are embedded in a 2-dimensional sheet and each has connections to a few neighbors. The first algorithm (**Promotion**) locates a leaf node physically near the failed node and links it into the tree as an in-place substitute. The second algorithm (**Chaining**) substitutes one of the tree-children of the failed node for its missing parent, then recursively substitutes at each lower level until a descendant which is a leaf and has no children is substituted for its immediate parent.

For both these algorithms, it is assumed that there are: f immediate children under each management node in the tree; h levels of nodes in each tree with the root at level 0; an expected standard physical distance d from each lowest level $h-2$ manager to its leaf children at level $h-1$; a time unit t needed for a message to travel a physical distance d ;

and a packing ratio r for the expected value of grandparent-to-parent versus parent-to-child physical distances at each level. For the purposes of this analysis, groups are assumed to be clustered in a plane, yielding $r^2=f$.

The general mode of analysis for each of the evaluated attributes of the algorithms is to determine the effect of a failure at an arbitrary level k , then multiply by the number of nodes at level k (f^k), and sum over all levels ($0 \leq k \leq h-1$). Dividing by the total number of nodes ($\frac{f^h-1}{f-1}$) determines the expected value for failures uniformly distributed throughout the network. For realistic tree fanout values near $f = 9$, each value is dominated by the results of leaf failures. The measures of each algorithm include:

- (1) **Impact**, the number of nodes directly affected by reconfiguration after one failure;
- (2) **Growth**, the increase in total tree branch physical lengths per reconfiguration;
- (3) **Traffic**, the product of the total number of messages times the physical distance each travels for all messages generated for one reconfiguration; and
- (4) **Delay**, the time for all reconfiguration messages to travel with the maximum parallelism allowed, during the recovery from each failure.

For d the short internode physical distance at leaf level, for t the unit time to send a message a distance d in the underlying physical network, and ignoring terms that are insignificant for large values of $N \approx f^{h-1}$, the simplified values are:

Algorithm Comparison Equations

Measure	Promotion	Chaining
Impact	$\approx 2 + \frac{2}{f^2}$	$= 2 + \frac{1}{f-1}$
Growth	$\approx d \left[\frac{r+1}{3r^3} \right]$	$\approx d \left[\frac{r+1}{3r} - \frac{1}{r^3} \right]$
Traffic	$\approx d \left[2r+1 - \frac{5(r-2)}{3r^2} \right]$	$\approx d \left[2r+2 + \frac{1}{(f-r)} \right]$
Delay	$\approx t \frac{2r^3+3r^2+r+2}{r^4}$	$\approx t \left[\frac{2(r+1)}{f} + \frac{2}{(r-1)f^2} \right]$

Assuming that fanout $f=9$, the values of the simplified expressions are:

Algorithm Comparison Values
(for Large Networks)

Measure	Promotion	Chaining
Impact	2.03	2.13
Growth	$0.05d$	$0.40d$
Traffic	$6.8d$	$8.2d$
Delay	$1.06t$	$0.90t$

The very simple **Promotion** algorithm is better both in terms of lessening expected branch growth and in terms of the total message traffic impact on the communications system. If all computing nodes in the network have equivalent computing and communication capabilities, the **Promotion** algorithm is preferred. If the network is non-homogeneous because nodes at the higher levels generally have better access to communications or other resources needed for effective management of the network, then the **Chaining** algorithm is preferred.

3. Conclusions

These studies are determining which distributed control algorithms can run efficiently on many large computing systems. The emphasis throughout is on evaluations for computing systems with thousands or millions of interconnected processing nodes. Massive parallel systems are becoming economically feasible and justifiable. An understanding of how to structure parallel algorithms for them is urgently needed.

4. References

1. W. D. Hillis, *The Connection Machine*, MIT Press, Cambridge, MA, 1985.
2. G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture", *Proceedings IEEE 1985 Intl. Conference on Parallel Processing*, August 1985, 764-771.
3. D. Gajski, D. Kuck, D. Lawrie and A. Sameh, "CEDAR - A Large Scale Multiprocessor", *Proceedings IEEE 1983 Intl. Conference on Parallel Processing*, August 1983, 524-529.
4. C. Maples and et al, "Performance of a Modular Interactive Data Analysis System (MIDAS)", *Proc. 1983 Intl. Conf. on Parallel Processing*, August 1983.
5. Tom Manuel, "Parallel Designs are Making Inroads", *Electronics*, 59, 10 (10 March 1986), 45-48.
6. Paul Walich and Glenn Zorpette, "Technology 86 - Minis and Mainframes", *IEEE Spectrum*, 23, 1 (January 1986), 36-39.
7. Alexander Wolfe, "Full Speed Ahead For Software", *Electronics*, 59, 10 (10 March 1986), 50-52.

8. Charles L. Seitz, "The Cosmic Cube", *Communications of the ACM*, 28, 1 (January 1985), 22-33.
9. John Beetem, Monty Denneau and Don Weingarten, "The GF11 Supercomputer", *IEEE Proceedings of 12th Annual International Symposium on Computer Architecture*, June 1985, 108-113.
10. Kai Hwang, "Multiprocessor Supercomputers for Scientific/Engineering Applications", *IEEE Computer*, June 1985, 57-73.
11. Jaap Hollenberg, "The Cray-2 Computer System", *Supercomputer*, July September 1985, 17-22.
12. Olaf Lubeck and James Moore, "A Benchmark Comparison of Three Supercomputers: Fujitsu VP-200, Hitachi S810/20, and Cray X-MP/2", *IEEE Computer*, 18, 12 (December 1985), 10-24.
13. Jerry Lyman, "Supercooling Comes to the Forefront", *Electronics*, 59, 10 (10 March 1986), 48-50.
14. D. H. Lawrie, "Access and Alignment of Data in an Array Processor", *IEEE Transactions on Computers*, C-24, 12 (December 1975), 1145-1155.
15. A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolf and M. Snir, "The NYU Ultracomputer - Designing an MIMD Parallel Machine", *IEEE Transactions on Computers*, C-32, 2 (February 1983), 175-189.
16. M. O. Rabin, "Probabilistic Algorithms", in *Algorithms and Complexity - New Directions and Recent Results*, J. F. Traub, (ed.), Academic Press, New York, 1976, 21-39.
17. L. D. Wittie and A. van Tilborg, "MICROS, A Distributed Operating System for MICRONET, A Reconfigurable Network Computer", *IEEE Transactions on Computers*, C-29, 12 (December 1980), 1133-1144.
18. L. D. Wittie and A. J. Frank, "A Portable Modula-2 Operating System: SAM2S", *Proceedings AFIPS National Computer Conference*, 53, (July 1984), 283-292.
19. C. Mohan and L. D. Wittie, "Local Reconfiguration of Management Trees in Large Networks", *Proceedings 5th Intl Conference on Distributed Computing Systems*, May 1985, 386-393.
20. A. J. Frank, L. D. Wittie and A. J. Bernstein, "Group Communication on Netcomputers", *Proceedings 4th Intl. Conference on Distributed Computing Systems*, May 1984, 326-335.
21. A. J. Frank, L. D. Wittie and A. J. Bernstein, "Multicast Communication on Network Computers", *IEEE Software*, 2, 3 (May 1985), 49-61.
22. A. J. Frank, L. D. Wittie and A. J. Bernstein, "Maintaining Weakly-Consistent Replicated Data on Dynamic Groups of Computers", *Proceedings 5th Intl. Conference on Distributed Computing Systems*, August 1985, 155-162.
23. A. J. Frank, "Distributed Dynamic Groups on Network Computers", PhD Thesis, Department of Computer Science, SUNY at Stony Brook, NY, August 1985.

THE DESIGN OF LOAD BALANCING STRATEGIES FOR DISTRIBUTED SYSTEMS

Rafael Alonso

**Department of Computer Science
Princeton University
Princeton, N.J. 08544
(609) 452-3869**

ABSTRACT

In this paper we consider the problem of designing and selecting load balancing mechanisms for distributed systems based on local area networks. In particular, we will focus on the type of information needed to make balancing decisions and on the desirable properties of the decision algorithms. We also describe our current approach to this problem, which essentially consists of carrying out a series of experiments on a prototype load balancing implementation. Finally, we present the insights we have derived from our experimental results.

1. Introduction

Many of the currently existing computing environments consist of a heterogeneous collection of workstations and mainframes connected by a high bandwidth local area network (LAN). One of the main benefits of working in a distributed system created for such an environment is being able to share scarce resources with other users of the network; but one resource that is often not shared is the processing capacity of the network nodes. In many systems, the scheduling of user jobs is individually carried out by each processor, and the computations of the users logged on at any one machine are performed locally. The decentralization of CPU management, coupled with large differences in the numbers (and types) of users connected to each of the nodes in the network, often can lead to situations where there are great disparities in load among the machines on the network. For example, at our local computer center, as the due date for a class assignment approaches, the processor assigned to the students in that class becomes heavily loaded, while other machines are underutilized. Although users can determine by themselves that an imbalance exists, and remotely log onto another computer, we feel that, in order to prevent a chaotic situation and to aid naive users, it is best to develop strategies that can solve the load balancing problem in an automatic way, much in the way that users now depend on virtual memory techniques to manage their memory space.

We are currently exploring the design of load balancing strategies for LAN-based distributed systems. In this paper, we describe some of the strategies that we are considering for implementation, as well as the criteria we are using to determine which schemes are appropriate for a given environment.

In the next section we provide some details concerning our approach to the problem. Sections 3 and 4 deal with load metrics and decision policies respectively (these terms are defined in Section 2). The last section of the paper discusses the current state of our work and presents some preliminary conclusions.

2. Our Approach

Although load strategies have been studied in the past, most of this work has been carried out within a theoretical framework (see [Chu1980] for a survey of some of these approaches); moreover, some of the researchers carrying out those studies have made simplifying assumptions that may not hold in practice. Also, there have been a small number of load balancing implementations (for example, see[Hwang1982]), but, as far as we know, the strategies employed were chosen without extensive study. Our work differs from most of the previous research in that, while we are interested in actually implementing a load balancing mechanism, we will not chose a strategy in an *ad hoc* fashion, but rather, we plan to devote extensive study to the choice of strategy to be programmed.

Before we describe our approach further, a few remarks are needed. A load balancing strategy is composed of two parts, a *load information* aspect (i.e., what information will be used to determine the load in the machines of the network), and a *decision policy* (i.e., given the load information, how will it be used to decide where to run a job). In order to speak meaningfully of a machine being "more loaded" than another, we define a *load metric* as a real-valued function of the load information. For example, the load information could be composed of the number of processes in the CPU ready queue for a given interval; a possible load metric would then be the average number of ready jobs during the last minute. This metric is essentially the "load average" metric provided by the UNIX 4.2 BSD [Leffler1984] **uptime** command. In the next two sections we consider load metrics and decision policies in greater depth.

Our current study of the load balancing problem comprises three phases. First, to consider what are suitable load metrics for our environment. Then, to list a number of possible decision policies that make use of the load metrics being studied. Finally, implementing various combinations of policies and metrics, in order to study their performance for a number of synthetic workloads.

We have already developed a prototype implementation of a load balancing mechanism, which can be used with different load strategies. It consists essentially of a shell that can make scheduling decisions based on load information broadcast by cooperating daemons (see [Alonso1986a] for more details). Our prototype now runs in a laboratory consisting of a variety of SUN workstations, connected by an Ethernet [Metcalf1976]. Although experiments are still being carried out, we will briefly comment on our current results in the concluding section of the paper.

3. Load Metrics

The selection of a load metric requires a careful definition of what is meant by the load of a processor. It seems clear that load should be defined, at least partially, in terms of a set of performance indices (such as CPU utilization or mean number of I/O requests), but it is less clear that two different processes should use the same definition of load; for example, an I/O intensive job will probably perceive load in a

different way than a CPU-bound job. Whether the benefits, if any, of using different load metrics for different tasks are sizable is not immediately obvious; at any rate, at present we are considering only global load metrics. Our rationale for this decision is twofold. In the first place, in our computational environment we typically do not know the characteristics of the jobs being executed, and thus, cannot compute job-specific load metrics. Secondly, we feel that a simpler technique has a better chance of working well in an implementation than a more complex approach.

A compromise between global and job-specific load metrics is to compute a different load metrics for each job class. The simplest such scheme would involve categorizing processes as either CPU or I/O bound. Although promising, this approach would not be of interest in our system, since most user jobs obtain their data (more precisely, all their pages) from a network file server. Thus, there is little one can do (in terms of migrating the task) for I/O bound jobs, except to exclude them from the load balancing algorithm. (Actually, this is equivalent to a load metric that has the same value for all machines under all conditions.) More will be said about the types of jobs that should not be migrated in the next section.

Another important issue is that, since load metric information cannot be constantly broadcast (because it would be too costly), the rate at which such broadcasts are made needs to be studied. The problem that arises is that machines are making their job scheduling decisions based on possibly stale data. This could lead to unstable behavior; for example, an idle node could start receiving migrated jobs from many other machines, and by the time it broadcast that it was overloaded, perhaps too many jobs would have been sent to it. We have already seen such behavior in our previous study of the possible improvements of introducing load balancing strategies in distributed database query optimizers [Alonso1986b].

Actually, the cost of frequent load information broadcasts involves two factors: the cost of computing the load metric, and the overhead of sending and receiving the messages over the network. One possible compromise would be to gather the load information more frequently, but broadcast it at less frequent intervals. The rationale for this is that, with this approach, each machine has a better idea of its local load, and can start migrating all of its jobs when it is overloaded. One of the results described in [Alonso1986b] suggests that, at least in the context of database jobs, this strategy can be quite successful. We plan to determine if that result holds for general purpose jobs.

A possible candidate for a load metric is the load average described above. Another is the *effective load average*, defined as the load average divided by the processor MIPS rate. A third is CPU utilization. The number of ready jobs plus the number of disks requests per unit of time seems a more comprehensive measure of load than a metric that involves solely the demand on the CPU. Another possibility is to compute the utilization of the bottleneck resource at every machine (since the throughput of a system is dominated by the scarcest resource, the utilization of this critical resource is a good indication of the node load). Lastly, a binary metric ("idle" or "not_idle") would be an appropriate choice if we were interested in migrating jobs primarily to unused machines on the network. Currently, we are planning to study most of the metrics just described.

There are a number of issues involved in evaluating a load metric:

- [1] The **stability** of a metric is important, because if it responds too quickly to minor changes in system parameters it may lead to instability in our load balancing strategies. The measurement technique used to compute the metric is relevant here (for example, exponentially smoothing a sample metric can improve the stability of the metric).
- [2] We are also interested in the **generalizability** of the metric, since we desire our schemes to be useful for non-UNIX systems as well as being valid for a heterogeneous set of hardware configurations.
- [3] Clearly, the **implementability** of the metric is of primary concern. Since we intend to carry out experiments using the various metrics, the ones that are either too difficult or too expensive to implement or to compute must be discarded.
- [4] Finally, there must be **empirical evidence** that the metric to be implemented actually reflects our intuitive notion of "load". This can be determined by experimental study.

Before we leave this topic it should be pointed out that, clearly, a load metric is only as good as the operating system statistics on which it is based. In the case of UNIX 4.2 BSD, it is not clear that the statistics provided by the kernel can be trusted completely. Thus, we also plan to carry out a study of the statistics gathering software of BSD UNIX.

4. Decision Policies

Before we describe some of the decision policies that seem attractive, we will touch upon some of the salient features of such policies. Policies can be categorized as more static or more dynamic; for example, always using the same computational server to offload a machine would be a very static policy, while choosing the server at run-time is more dynamic. Some policies require knowing varying amounts of information about a process in order to schedule it (e.g., a strategy may require that we know if a process is CPU or I/O bound). The scheduling decision may be made at a central site or in a distributed fashion (we are only interested in the latter); also, the location of the decision mechanism may impact the number of messages that have to be sent to communicate load information. A key feature of any policy is whether it allows preemption or not (i.e., whether we are allowed to keep migrating jobs after they start executing); we will not consider such policies since we cannot currently implement them in a UNIX environment. Some policies may limit the processors that may be chosen; a possible strategy may be to let only idle nodes compete for tasks, or to consider only machines with a certain MIPS rate for task allocation. The choice of sender-initiated versus receiver-initiated balancing must be settled (i.e., whether to allow busy nodes to look for an idle node, or to have idle nodes advertise their ability to work). Finally, it is clear that some jobs should never be migrated; this may be so for a variety of reasons: perhaps the jobs should only be run locally for security reasons, or maybe the computational demands of the task are so slight that the overhead of moving it overshadows any possible performance gain. There are still many other issues to be considered: do we optimize for the current task or do we look at sets of tasks? do we examine only jobs that have arrived at our local decision maker

mechanism or at all the incoming jobs in the network? do we remember previous task assignments when we make a decision? which performance index do we focus on?

There are many policies that could be of practical use. Perhaps the most natural policy is to send jobs to the "least loaded" machine. A variation that may prove more stable consists of migrating jobs to any machine whose load is less than the local load by a specified amount. In order to decrease the chance that all busy machines select the same host to which they will migrate their tasks, each machine could choose the destination randomly from the n least loaded processors (for some n). Another alternative is to only move jobs to idle processors. Finally, always sending the jobs of overloaded processors to a powerful computational server may be reasonable for some environments.

In judging a decision policy we may consider a number of factors:

- [1] As we mentioned above, the **stability** of the policy under imperfect information is important, since we expect that imperfect information will be the rule rather than the exception.
- [2] We would prefer that the **cost** of the load balancing scheme be negligible compared to its benefits (note that there are two types of costs, the costs to users of using the system, and the cost to non-users of the added overhead of running the decision mechanism).
- [3] Also, the load balancing scheme must not force all the processors in the network to guarantee a given level of service; in particular, some processors may refuse service or allow only certain classes of tasks to be assigned to them. Thus, the amount of **autonomy** that machines have under a given policy is important. (Note that the receiver-initiated policies mentioned above seem to simplify achieving this goal.)
- [4] Finally, the amount of **transparency** in the load balancing scheme is also of critical importance, since it would be desirable to maintain users unaware of the fact that their jobs may be running remotely.

5. Conclusions

In this paper we have presented our ideas concerning the implementation of load balancing mechanisms. We have pointed out possible load metrics as well as sample decision policies, and have detailed some of the relevant issues involved in making a choice from the many possible strategies. It seems clear to us that the selection of a load balancing strategy involves many complex choices which require careful study, and it is certainly not clear *a priori* which strategy to pursue.

At the present time, we are actively involved in a series of experiments that explore some of the ideas presented in this paper. Our previous work ([Alonso1986a] and [Alonso1986b]) suggests that the gains due to load balancing can be quite sizable, and our current results continue to be promising. Moreover, we feel that this area of research contains many problems of both practical and research interest and merit further study.

References

Alonso1986a.

Alonso, Rafael, Goldman, Phillip, and Potrebic, Peter, "A Load Balancing Implementation for a Local Area Network of Workstations," *Proceedings of the IEEE Workstation Technology and Systems Conference*, March 18-20, 1986.

Alonso1986b.

Alonso, Rafael, "Query Optimization in Distributed Database Systems Through Load Balancing," Ph.D. Dissertation, U.C. Berkeley, 1986.

Chu1980.

Chu, W. W., Holloway, L. J., Lan, M., and Efe, K., "Task Allocation in Distributed Data Processing," *IEEE Computer*, November 1980.

Hwang1982.

Hwang, K., Croft, W. J., Goble, G. H., Wah, B. W., Briggs, F. A., Simmons, W. R., and Coates, C. L., "Unix Networking and Load Balancing on Multi-Minicomputers for Distr. Proc.," *IEEE Computer*, April 1982.

Leffler1984.

Leffler, S., Joy, W., and McKusick, K., *4.2 BSD System Manual*, Computer Systems Research Group, University of California, Berkeley, 1984.

Metcalfe1976.

Metcalfe, R. M. and Boggs, D. R., "Ethernet: Distributed Packet Switching for Local Computer Networks," *CACM*, vol. 19,7, pp. 395-404, July 1976.

Experiments with Parallel Program Execution on a Network of Workstations

Willy Zwaenepoel*
Department of Computer Science
Rice University
Houston, TX 77001

March 14, 1986

Abstract

In this project we study the parallel execution of programs on a collection of workstations connected by a local area network. So far, our work has focused on developing a number of pilot applications, including a distributed Pascal compiler and a distributed *make* facility. We intend to use these applications as an experimental testbed for studying issues in problem partitioning, fault tolerance, machine allocation, and monitoring and debugging of distributed programs.

1 Introduction

In recent years there has been an evolution away from centralized timesharing systems towards systems consisting of workstations connected by a high-speed local area network to specialized server machines. The main advantage of workstation-based systems is that each user has the full power of a dedicated machine at his disposal. Therefore, performance depends less on the overall number of users of the system.

Unfortunately, this advantage does not come for free. In particular, most workstation-based systems do not allow convenient access to the computing power of other machines in the system, even if those machines are idle. Centralized systems, in contrast, can typically make all of their idle resources available to a single user. This weakness of most workstation-based systems is particularly serious when one realizes that demand for computing power is highly bursty making strict preallocation of resources highly inefficient. For instance, when compiling a large collection of programs with the UNIX *make* facility, one could easily use the power of many workstations if the compilations can be run in parallel. Nevertheless, one is essentially restricted to the computing power of one's own workstation, regardless of the fact that a large amount of computing power on other machines may be available.

This argument is not to be interpreted as one against workstation-based systems. Due to the economics of VLSI and local network technology, it is currently more cost effective to build ten 2-MIPS machines and connect them by a suitable high-speed network than to build

*This research is supported in part by NSF Grant No. DCR-8511436 and by an IBM Faculty Development Award.

a single 20-MIPS machine. Also, sharing a single machine entails a substantial amount of overhead for providing protection and fair sharing among users: a 20-MIPS machine shared by ten users often appears less powerful than ten dedicated 2-MIPS machines. Nevertheless, we see potential in using a collection of workstations in a more multiprocessor-oriented manner. Such an arrangement is especially attractive if the workstations are diskless, thereby avoiding the need to worry about the owner's local private storage.

We are using the V-System developed at Stanford University as the base tool for our research [5, 4, 2]. The V-System is a relatively complete distributed operating system for a collection of SUN workstations connected by a 10-megabit Ethernet. It consists of a kernel resident on each machine in the system and a collection of server processes. The kernels cooperate to provide transparent (location independent) communication between processes. The server processes provide the other services typically thought of as part of the operating system, such as file access and network access. Additionally, a large number of application programs have been written for the system, including a window system, text editors, a graphics editor, and compilers for C and Pascal. Much emphasis in the design of the V-System has been placed on high-speed interprocess communication. We believe highly efficient (network) interprocess communication is essential if we are to investigate the limits of the granularity of the decomposition that we can afford in a network environment.

The project described here was started in January 1986 with funding from the National Science Foundation [1]. So far, we have concentrated on developing a small set of prototype parallel programs. In particular, we have developed several approaches to parallel compilation [3, 7] as well as a parallel implementation of the *make* program. These applications are described in Section 2. While we continue to refine these applications we also intend to use them as a testbed for investigating issues such as fault tolerance, machine allocation, and debugging and monitoring of distributed programs. Our initial approaches to these problems are described in Section 3.

2 Pilot Applications

2.1 An Experimental Parallel Compiler

A compilation task is composed of a number of phases including scanning, parsing, semantic analysis, code generation, optimization and possibly assembly. Our approach to speeding up compilation is based on the observation that scanning and parsing should take relatively little time compared to the other phases of the compilation process. As a result, scanning and parsing are done sequentially, while the other phases are executed in parallel. Given this observation, we need to find a framework for expressing semantic analysis and code generation that is amenable to parallel execution. We use *attribute grammars* for this purpose. In an attribute grammar specification, each nonterminal is annotated by a set of attributes. *Semantic functions* specify the values of the attributes of nonterminals in a given production in terms of the values of other attributes of nonterminals in the same production. From a language viewpoint, the strength of attribute grammars result from the fact that they provide a high-level, non-procedural definition of the semantics of a programming language. Efficient implementation of an attribute grammar evaluator on a sequential machine is non-trivial, though. When considering a parallel machine, however, the applicative nature of an attribute grammar specification lends itself particularly well to efficient parallel implementation. This applicative nature leaves the order in which attributes are evaluated relatively unconstrained, and as a result, synchronization overhead is kept to a minimum.

We have built a prototype implementation of this system for a sizable Pascal subset. A sequential parser builds the syntax tree and sends out subtrees for evaluation to a set of parallel evaluator processes. Each of those processes evaluates the attributes of its subtree and communicates the attribute values with other evaluator processes. Finally, the root evaluator sends back the "code" attribute to the parser. Preliminary experiments with this prototype show a speedup factor of 3 to 4 for 8 processors (the maximum currently available for our experiments), with sequential execution speeds comparable to conventional compilers [3].

Much work remains to be done on this specific problem. First, we need to get a better idea of the behavior of the parallel compiler in terms of computation, communication and synchronization. Second, the algorithm used in the evaluator is a straightforward *dynamic* algorithm. State-of-the-art sequential evaluators use a *static* evaluation algorithm whereby dependency analysis is done at generation time and not at execution time. We wish to explore the combination of static and parallel evaluation. Finally, a full implementation of a language relevant to our research is desirable so that we can use the parallel compiler in our daily work.

2.2 A Parallel Make Facility

The standard UNIX *make* facility automates the compilation of source code files (and other types of processing) by allowing the programmer to specify the dependencies among the various files that make up a program and record instructions about how the results are to be built. In the traditional, sequential version of *make*, the system builds one target at a time. As a result, complex targets that require the execution of many commands require a large amount of time. Since a large number of these commands tend to be independent (for instance, compilations of unrelated modules), many of these commands can be executed in parallel. Parallel execution is accomplished by sorting the task dependency graph in topological order and starting up as many commands in parallel as possible within the constraints of the dependency graph. The main "controller" process awaits completion of the subtasks, updates the dependency graph accordingly, and starts up new commands that have become ready.

The speedup resulting from parallel execution is very dependent on the actual dependency graph. A single large compilation combined with a number of small ones tends to produce only moderate speedups, even with large numbers of machines available. On the other hand, if a large number of compilations of approximately equal length are required, nearly linear speedup can be achieved with a large number of processors. Early experiments have also shown very significant loads on the file server as a result of a parallel *make* (all of our workstations are diskless). This suggests the need for improved caching strategies on the workstations and, perhaps, the need for parallel program loading if a broadcast network is used.

3 Research Issues

3.1 Fault Tolerance

In recent years, a number of fault-tolerance mechanisms for distributed systems have been developed. However, most of these mechanisms either restrict the types of applications that can be run (for example, requiring them to be structured as a series of atomic transactions) or require specialized hardware to support the communication between machines needed for the fault tolerance. Also, these mechanisms tend to place a large amount of overhead on the system in both increased computation and communication costs. What we would like is a simple,

low-overhead technique to achieve fault tolerance that can be used with existing distributed applications programs and standard workstation and local network hardware.

A popular method of achieving fault tolerance is to periodically checkpoint each process (individually) and to log all message traffic which a process receives either on disk or with some special logging process on a separate machine. Then, if a process fails, it can be restarted from its most recent checkpoint, and the messages which have been logged since that checkpoint can be replayed to it, thus restoring its internal state to the point at which the fault occurred. The frequency of checkpointing can be tuned to balance the expense of the checkpoint operations against the time needed for recovery, but the logging must be done for each message that each process receives, placing a continuous overhead on the operation of the system even in the case where no faults ever occur.

The solution we are exploring is based on an analysis of the minimum-cost method of accomplishing this logging [6]. Since the sender and receiver processes get a copy of each message exchanged between them, it would be convenient if one of them could serve as the log by saving a copy of the message in its own local memory. Unfortunately, the receiver can not log the message in this way since the purpose of the logging is to be able to recover if the receiver fails. The sender, though, could log the message, except that it is also necessary to log the order in which messages were received which the sender does not normally know. We believe that by having the receiver reply back to the sender with a message ordinal number, we can have the necessary recovery information available at very little cost in performance.

3.2 Machine Allocation

Allocation of computational tasks to processors in a multi-machine environment has received a great deal of attention, especially in theoretical work. However, relatively little experience exists with real implementations. We hope to provide some experimental evidence on the relative merits of different techniques that have been studied in a theoretical framework.

Given our environment, the problems of machine allocation are compounded by a number of extra complications. First, the machines are not dedicated to compute-intensive work. In first order, these machines should provide highly responsive interaction to their owners. These "guest" computations should not be allowed to degrade interactive response, but at the same time the execution time of the compute-bound jobs should be minimized. Second, due to the relative abundance of CPU power when all machines are combined and the fact that compute-bound tasks are generated rather sporadically, many of the common assumptions made in previous studies do not seem to hold. For instance, CPU run queue length might not necessarily be a very valuable indicator of load on a workstation, since it usually will be either 0 or 1. There also seems to be little correlation between present and future load, unlike on timesharing systems where the large number of users smooths out load fluctuations.

3.3 Monitoring and Debugging

Most of the measurements on our current distributed programs are done by *ad hoc* techniques. Counters are inserted in selected places in the program, and network-wide monitors are occasionally useful in getting system-wide measurements. More sophisticated tools for debugging and performance monitoring are essential, however. We are interested in getting two kinds of measurements. First, we would like to get a better characterization of local network traffic in an environment of diskless workstations, especially when those workstations are heavily used for parallel program execution. Older studies done in a less integrated environment indicate that

the load on a local network tends to be very low. In an environment of diskless workstations where all secondary storage is accessed over the network and where programs execute on different machines, this evaluation needs to be reassessed. Second, we are interested in exploring whether there is a way to combine communication traces with more traditional techniques for debugging distributed programs.

4 Acknowledgements

The research group involved in this project consists of two faculty members (Guy Almes and the author) and four graduate students (Rick Bubenik, Jerry Fowler, David Johnson and Allan Porterfield). Hans-Juergen Boehm has been a key contributor to the parallel compiler effort.

References

- [1] G. T. Almes and W. Zwaenepoel. *Understanding and Exploiting Distribution*. Technical Report Rice COMP TR85-12, Department of Computer Science, Rice University, February 1985.
- [2] E. J. Berglund et al. V-System reference manual. Computer Systems Laboratory, Stanford University.
- [3] H. J. Boehm and W. Zwaenepoel. An automatically generated parallel attribute grammar evaluator for a loosely coupled multiprocessor. Department of Computer Science, Rice University.
- [4] D. R. Cheriton. The V kernel: a software base for distributed systems. *IEEE Software*, 1(2):19-42, April 1984.
- [5] D. R. Cheriton and W. Zwaenepoel. The distributed V kernel and its performance for diskless workstations. In *Proceedings of the Ninth Symposium on Operating System Principles*, pages 129-140, ACM, October 1983.
- [6] D. B. Johnson. Low-overhead fault tolerance mechanisms for distributed systems. Thesis Proposal (in preparation), Department of Computer Science, Rice University.
- [7] D. B. Johnson and W. Zwaenepoel. Macropipelines on a network of personal workstations. Department of Computer Science, Rice University.

Session 11: Distinguished Lecture

by

Stephen Lundstrom

Microelectronics and Computer

Technology Corporation

Austin, Texas

Chairperson:

Dharma P. Agrawal

North Carolina State University

Future Directions in Computer Architecture and Software - The Year 2000

Dr. Stephen F. Lundstrom
Vice President and Program Director

Parallel Processing Program
Microelectronics and Computer Technology Corporation
Austin, Texas 78759

Preface

At the request of the National Aeronautics and Space Administration, the National Research Council's Aeronautics and Space Engineering Board conducted a workshop in January 1984 to project what the state of knowledge in aeronautical technology could be in the year 2000, if necessary supporting resources were made available. Some 80 experts participated in the study. They were organized into eight panels in the areas of aerodynamics; propulsion; structures; materials; guidance, navigation, and control; computer and information technology; human factors; and systems integration.

This author chaired the panel on computer and information technology. This paper provides a brief summary of the major findings of this panel together with some more recent observations. The full report of the panel can be found in the report **Aeronautics Technology Possibilities for 2000: Report of a Workshop** (available from the Aeronautics and Space Engineering Board, Commission on Engineering and Technical Systems, National Research Council, 2101 Constitution Avenue, N.W. Washington, D.C. 20418). A copy of the report of the computer and information technology panel is included with this paper as an Appendix. A paper based on this report, **Computer and Information Technology in the Year 2000 - A Projection**, by Lundstrom and Larsen is available in *IEEE Computer Magazine*, September 1985, Vol. 18, No. 9, pp. 68-79.

August 14, 1986

Future Directions in Computer Architecture and Software - The Year 2000

Dr. Stephen F. Lundstrom
Vice President and Program Director

Parallel Processing Program
Microelectronics and Computer Technology Corporation
Austin, Texas 78759

Computers are playing significantly larger roles in many application domains today. This growth is expected to continue since information processing technology has advanced by a factor of 10 every 5 years for the past 35 years and is expected to continue to do so. Breakthroughs in device technology, from vacuum tubes through transistors to integrated circuits, contribute to this rapid pace. This progress is nearly matched by similar, though not as dramatic, advances in numerical software and algorithms. Progress has not been easy. Many technical and non-technical challenges were surmounted. The outlook is for continued growth in capability, but will require surmounting new challenges.

The technology forecast presented in this paper was developed by extrapolating current trends and assessing the possibilities of several high-risk research topics. In the process, critical problem areas that require research and development emphasis have been identified. The outlook assumes a positive perspective; the projected capabilities are possible by the year 2000, and adequate resources will be made available to achieve them.

The following summary of the outlook to the year 2000 is taken from the Computer Magazine article cited above.

The capabilities, on the hardware side, will be based on components that are 100 to 1000 times more cost-effective and size-effective than those available today, as well as on components with major new capabilities such as arrays of solid-state sensors. Airborne systems will be synthesized from heterogeneous collections of processors, some of which will execute at rates up to 1,000 MIPS and will contain 50 Mword of random access memory. Product designers will have the use of workstations (with local processing rates of 100 to 1000 MIPS on 32-bit operands, with 50 Mword RAM, and rotating storage of 10 to 100 Gwords) and supercomputers (composed of 100 to 100,000 processing elements, with cumulative processing power of up to 1000 Gflops and containing up to 10 Gwords of RAM). These various systems will take advantage of software that will support distributed, heterogeneous databases, real-time flight management and crew assistance, as well as multidisciplinary design and manufacturing support tools.

Computer and information technology is supporting and enabling technology to the aeronautics industry *as well as many others of course*, both directly and indirectly. The technologies considered by the panel were those related to application within aeronautics products directly, to research leading to the development of technologies needed for future aeronautics products, to the design, development, and manufacturing of the aeronautics products, to the support of these products including maintenance, diagnostics, and provisioning, and to the use of the products, including support of the impact on the flight crew. The benefits of computing are realized (indirectly, usually) through disciplines such as aerodynamics, and guidance, navigation and control. Therefore, the impact of computing on aerospace technology *or technology in other areas* will be in direct proportion to the extent to which these *supporting disciplines exploit computing*.

Assimilation of the technology into engineering practice will continue to be a major challenge. The critical technical impediments will be trustworthiness, size/performance, and management of complexity. The major challenges are likely to come from the areas of software development, system verification and validation, communication and control, and programs used to augment human intelligence, such as expert systems, theorem proving, and symbolic mathematics.

Possible Impact of Advancing Technology

The following paragraphs are based on information obtained since the report summarized above was prepared.

Engineering/Manufacturing Infrastructure

A major problem in many industries is the shortening product life cycle combined with future products which are more complex and seem to require more time and resources for their development. A dramatic reduction in development costs is critically needed. Major reductions in development costs would include significantly reduced development time, allowing first delivery of a product to be much earlier in the product life cycle. Future computer and information technology is expected to play a major role in accomplishing such a reduction. Some of the specific areas where major contributions can be expected are:

1. Rapid Turn-Around Design,
2. Flexible Manufacturing,
3. Integrated Vendors, and
4. Micro Manufacturing.

All of the above factors relate to reducing product cost, either by reducing product development costs, or by reducing the cost of manufacturing. In addition, all of these factors will be enabled by advancing computer and information technology. Consider each of these briefly:

Rapid Turn-Around Design Computer and Information Technology will significantly change the way in which complex designs are approached. New means for groups of people to efficiently interact during the design process will be developed. The computer-based tools will significantly enhance the individual's capacity to manage complexity. As a result, product design will require significantly less manpower for a given level of complexity, leading to reductions in the product design cost and to reductions in the time needed to complete a design. Computer tools will enable the implementation of special purpose hardware as easily as software can be developed. These advances should reduce the time to introduction of a product, thus reducing the cost commitments to the development of the product before cash return can be realized.

Flexible Manufacturing The advent of general purpose manufacturing handlers and tooling will allow the same manufacturing facilities to be utilized in the production of many different products, simultaneously. An example of this is how compiler technology is being used to develop such a capability in the laboratories at the Center for Integrated Systems at Stanford University. Dr. Brian Reid has been directing the development of a new language, called FABLE. FABLE is an integrated circuit manufacturing process specification language. When a designer has completed the design of an integrated circuit, both the mask set (or mask set definition), and the specification of the manufacturing process are released to the semiconductor "foundry". The mask set controls the patterning on the integrated circuit device being produced. The FABLE process specification is compiled into modules which control the various production facilities in order to exactly duplicate the desired manufacturing conditions. The concept of silicon foundries, where one vendor provides a manufacturing service to a designer, can be expected to be utilized well in other manufacturing areas as well.

Integrated Vendors Both the design and manufacturing process of large, complex systems often require the use of vendors (both during design and manufacturing). If the computer tools (in

design, manufacturing, and accounting) of both the manufacturer and the vendors are appropriately interfaced, the information flow between companies (now handled with slow manual procedures) could be streamlined with a resulting reduction in the overall time for product development. If the manufacturer and vendor computer systems are integrated appropriately, significantly fewer errors would be expected since all parties involved would have exactly the same information, with much less chance for human error in the dissemination and coordination of the information.

Micro Manufacturing Once manufacturing facilities are automated and generalized sufficiently that a broad range of manufacturing duties can be accomplished on the same equipment, small flexible factories can be expected to be created near the major centers of demand. These small, flexible factories would receive specifications of what to manufacture through an electronic ordering process. Finished goods would already be near the customer, with a significant reduction in shipping costs.

Impact of Non-Technical Decisions

One of the challenges of any long-range forecast is to second guess what the actual motivating conditions will be to actually drive the development of technology. These conditions were studiously avoided in the report summarized earlier. Following is a brief discussion of three possible motivating conditions and their impact on the advance of computer and information technology.

1. Oil Crisis
2. Accounting/Office Management
3. Factory Automation

A sustained oil crisis would probably result in neighborhood centers containing offices of nearby residents, micro factories, child care centers, post office, neighborhood shops, etc. Locally centered life would not only reduce the consumption of oil, the requirements for interconnected offices and businesses would grow substantially. Community center workplaces and micro factories require significant advances in distributed computing as well as security so that the many companies represented in such a heterogeneous workplace could feel confident that their portion of the workplace is free from spying eyes.

As the cost of computing plummets even faster, the traditional tradeoffs between the cost of employees and the tools which they use changes rapidly. The productivity of people in the workplace, even in many office management and administrative roles, will become ever more important. Computer and information technology will be used to significantly improve the productivity of these workers.

The factory automation necessary to implement the flexible factories and the micro factories will likely cause displacement of workers. This rapid change can be expected to encourage the development of a new, computer-based education industry to serve the needs to rapidly retrain displaced workers.

The Future - 2000 and Beyond

A new computer organization, inductive machines, may be available in the future. This term, coined by Lipovski and Malek at the University of Texas, refers to a system which can begin with a functionally complete system containing at least one of each subsystem. The system can then be expanded by adding a copy of the initial system with the resulting system acting the same as the basis system, but faster. As more and more speed is needed, additional copies of the initial basis system are added. At present, such an inductive organization seems feasible, although the system's speed may not increase as fast as the costs when the size is increased. However, the existence of inductive applications is not yet clear.

In today's terminology, we say that a computer "solves" our problem. However, to be more precise, we start with a problem, we determine how to solve the problem, we prepare a set of precise instructions to a computer which, when executed, will implement the solution which we invented. With the advent of complex expert systems in the future, users should be able to approach a computer with a description of their problem and expect the computer to determine how to solve it. We might say that such a computer is "problemmed" rather than "programmed".

When trying to determine what markets will motivate developments in the future, one can identify many new, potential uses of computers and systems which they will be a part of. We should not forget, however, that businesses will *still* be doing accounting, with this possibly being the most prevalent application of all since each business, and possibly each worker would need to be using such services.

A last mention about an area often overlooked in discussions of future directions in computing, that of real-time processing. Real-time applications historically have not attracted the attention of researchers interested in the development of tools and architectures suited to such systems. Applications are predominately developed using assembly language because the time cost of each instruction in a program is easily identified. However, once highly concurrent systems are available, even the use of assembly language will not be sufficient to determine, a priori, the actual flow of execution across time. New tools, from language constructs to define time orders, constraints and dependencies in applications to time constraint enforcers in future systems, are badly needed. Enough research is now beginning that these needs may also be addressed reasonably well by the start of the twenty-first century.

Session 12: Concurrency Control

Chairperson: John Zavada
U.S. Army Research Office

DISTRIBUTED QUERY PROCESSING - PRESENT STATUS AND FUTURE DIRECTION

C. T. Yu and K. C. Guh

Department. of Electrical Engineering and Computer Science
University of Illinois at Chicago
Chicago, Illinois 60680

1. Summary

The aim is to construct a powerful distributed query processing system, which is suitable for local area networks, long-haul networks and mixed networks where local area networks are interconnected by long distance lines. The system will be capable of processing logical queries (e.g., Prolog) as well as ordinary relational queries (e.g., Ingres types of queries), be able to learn to process queries more efficiently through being use and be able to adapt itself to varying environments (e.g., partial system crash survivability).

2. Present status: Semi-join algorithm and replicate algorithm

Distributed query optimization is an important factor for the performance of a system with databases distributed in a network. Many distributed query processing algorithms [ApHY, BeCh, BeGo, CDFG, ChHo, BiLu, ChBH, Chan1, Chan2, ChLi1, ChLi2, ChLi3, EpSW, GoSH, HeYa, Luk2, SDD1V1, SDD2V2, WoKa, Wong, WOHL, Yaos, YCTB, YLCC, YuOz] have been proposed. Most of them can be classified into two categories. The first category, semi-join algorithms [ApHY, BeCh, BeGo, CDFG, ChHo, BiLu, ChBH, Chan1, Chan2, ChLi1, ChLi2, ChLi3, GoSH, HeYa, Luk2, SDD1V1, SDD2V2, WoKa, Wong, WOHL, Yaos, YCTB, YLCC, YuOz], emphasizes on the reduction of the amount of data transferred across sites in networks. These algorithms are suitable for databases distributed in long-haul networks. The other type of algorithms, replicate algorithms [EpSW, YGCC, YGZT, TBCD], emphasizes on the reduction of local processing cost by promoting parallel processing in different sites, though communication cost is taken into consideration. These algorithms are suitable for databases distributed in local area networks. We will briefly describe what we have accomplished in these two categories. Our algorithms [YCTB, YGCC, YGZT] are intended for ad hoc queries, unlike enumerative algorithms ([ChHu, LMHL, WOHL]) which are more suitable for repetitive queries. Standard relational terminology can be found in [Date, Ullm].

2.1. Semi-join algorithm

A semi-join from relation R_i to relation R_j on attribute A [BeCh], denoted by $R_i -A \rightarrow R_j$, is defined to be the join of R_i with R_j on A projected back onto the attributes of R_j . If R_i and R_j are in different sites, the semi-join operation can be performed by projecting R_i on A , sending the resulting unary relation to the site containing R_j and then joining with the relation R_j on A . In this way, the amount of data transfer will be smaller than that of sending the entire relation R_i . Furthermore, the semi-join

operation eliminates tuples of R_j that do not have a common value with R_i projected on A . Thus, the operation always reduces the right-hand operand, so that if it is to be transferred to another site, less data transfer will be required.

The algorithm has been reported in [YCTB, YCTBL]. It is similar to many existing distributed query processing algorithms [ApHY, BILu, Chan1, Chan2, ChHo, ChLi1, ChLi2, ChLi3, HeYa, SDD1, etc.] in executing a sequence of semi-joins [BeCh] to reduce the sizes of relations referenced by the query, before assembling them in a site to obtain the answer. However, it has the following additional desirable features: (i) The algorithm determines the relations that need not participate in later operations after the execution of some semi-joins; as a result, communication and local processing costs due to the processing of those relations are saved. (ii) All semi-joins involving relations identified in (i) can either be discarded or replaced by better semi-joins; as a consequence of (i) and (ii), the algorithm can be shown to yield better performance than the SDD α algorithm [YCTB1, YCTBL]. (iii) For each semi-join involving horizontally fragmented relations, it determines a strategy with least communication cost to execute the semi-join. (iv) It makes use of redundant copies of relations to reduce communication cost and promote parallelism. (v) Semi-joins are executed dynamically (i.e. as soon as a semi-join is planned, it is executed and the number of tuples of the reduced relation is returned) so that the errors in estimating the cost and the benefit of executing the next semi-join are reduced. Furthermore, if execution errors or site failures are encountered, alternate strategies may be invoked.

2.2. Replicate algorithm

The replicate algorithm [YGCC, YGZT] makes use of semantic information so that certain distributed queries can be processed locally without data transfer with respect to the join clauses of the query. A simple algorithm is given to recognize the "locally processable queries". For non-local processable queries, a simple "fragment and replicate" approach [EpSW] is used. The approach chooses a relation to remain fragmented while replicating the other relations referenced by the query at the sites of the fragmented relations. However, unlike [EpSW], our algorithm takes into considerations not only the amount of data processed and transferred but also the presence or absence of fast access paths (e.g. indices), which have a very significant effect on local processing cost, and different processing speed at different sites. If the fragments of the chosen relation are not moved, then a linear time algorithm that minimizes the response time can be found; otherwise, the problem can be shown to be NP-hard [YGCC]. In the latter situation, heuristics are suggested. Experimental results given in [YGZT] show that the heuristics give good approximation. Details can be found in [YGCC, YGZT].

When all the relations referenced by a query are unfragmented, it may be desirable to partition a relation into fragments before the replicate strategy is applied. An optimal algorithm to choose the partitioned relation and a set of processing sites is given in [YGBC].

3. Future directions

We believe that the semi-join algorithm and the replicate algorithm should be modified to allow (a) integration, (b) adaptation and (c) be capable

of processing logical queries.

(a). Integration

Both replicate algorithm and semi-join algorithm suffer from being too restrictive in the sense that they are suitable for only a particular network environment (i.e., local area networks or long-haul networks) and do not take advantage of the features of one another. It is believed that a semi-join operation is useful in reducing not only communication costs but also local processing costs [VaGa]. Furthermore, if a relation can be eliminated after executing some semi-joins [YCTB], the chance of lowering further local processing costs and communication costs is greatly enhanced. We are developing an integrated algorithm (integrate semi-join and replicate algorithms) [YuGC]. The feature of the algorithm is briefly described as follows.

The concepts of cost, benefit and profit (benefit - cost) of executing a semi-join is also used as in other semi-join algorithms for long-haul networks (e.g., [SDD1, YCTB]), except they are generalized in two ways. First, local processing cost as well as data communication cost are incorporated. Second, benefits and profits are defined at each processing site (as determined by the replicate algorithm). We also introduced the concept of the relative profit. Let Res be the response time and $Total$ be the total cost (communication costs and local processing costs) at a site before performing a semi-join. The relative profit at this site due to performing a semi-join is defined to be $Res - Total + profit$. The smallest relative profit among those of a semi-join at all processing sites is defined to be the minimum relative profit of the semi-join. If the minimum relative profit of a semi-join is greater than zero, then the response time can be reduced by executing the semi-join. In other words, for those sites which have total costs close to Res before executing the semi-join, we need profits > 0 to find the semi-join worthwhile executing. But for those sites which have total costs much less than Res , though the profits may be less than 0, the semi-join is still worth executing as long as its final response time $< Res$. The extension of the concept of the minimum relative profit of a semi-join provides a vehicle to select semi-joins. The semi-join having the largest minimum relative profit among all possible semi-joins is chosen first, then the process is repeated to choose the next semi-join having the largest minimum relative profit and so on. In this way, semi-joins are incorporated into the replicate algorithm.

The integrated algorithm is to be used not only in local area networks or long-haul networks but also in mixed networks. If a query can be processed within a single local network, then the integrated approach given above is likely to yield a good strategy. Suppose that a query has different fragments spanning a number of local networks. Each query fragment may be processed using the integrated replicate algorithm. Then the query unifying the different query fragments may be processed using the semi-join algorithm. Whether this is a reasonable approach has yet to be investigated.

(b). Adaptation

A powerful query optimizer should be able to yield not only a good strategy for processing a query but also adapt itself to varying environments by acquiring knowledge from executing previous queries.

There are a number of situations in which the experience obtained in the execution of queries will be useful in speeding up the execution of future queries. Some of these are given as follows.

- (i) After executing a semi-join of the form $R_i \rightarrow A \rightarrow R_j$, if we find that R_j is not reduced, then $R_j(A)$ is known to be a subset of $R_i(A)$. This allows us to save in the processing of certain queries containing join clauses of the form $R_i \cdot A = R_j \cdot A$. More information can be found in [YLGT].
- (ii) The cost of executing a step in each of our two algorithms (the replicate and the semi-join algorithms) is estimated. When the step is executed, the actual cost is returned due to the dynamic nature of our algorithms. If there is a significant difference between the estimated cost and the actual cost, the estimated cost is increased or decreased by a fraction of the difference between the two costs. As an example, suppose the step is to execute a data transfer in a long-haul point-to-point network and the actual cost is a lot higher than the estimated cost. Then the phenomenon may indicate that the traffic between the two sites is congested. The estimated cost will be increased and as a result, data transfer between the two sites in the near future may be replaced by data transfer involving different sites if the alternative is acceptable and cheaper. Similarly, if the step is the execution of some local operations at a site and is found to be a lot more expensive than estimated, then the site may be overloaded. With the estimated cost increased, equivalent operations will automatically be transferred to other sites so as to achieve load balancing [CaLL,Ston]. More details can be found in [YLGT].
- (iii) It is possible that there are some situations in which the algorithms may perform poorly, especially in a mixed network environment. Some users, having certain local knowledge may be able to provide strategies for some queries with better performance than our algorithm. We have designed a graphical interface [YLGW] in which users can specify their strategies by stating the sequence of semi-joins to be executed, or the relation that is to remain fragmented and their processing sites, or some combination of the two. The user strategy is executed. If the actual cost of the user's strategy is significantly less than the estimated cost of the system's strategy, then the user will be asked to explain the basis of his/her strategy. The user's strategy will also be saved by the system for later examination. In some situations, an automatic analysis is performed and the user is not required to provide the reasons for his/her strategy. It is hoped that knowledge which is not exploited by our algorithms will be unveiled by either experienced or lucky users and then the knowledge will be acquired by our algorithms.

In addition, query optimization algorithms in highly available systems (e.g. [Kim, BhLi]) must be able to adapt to the current status of the sites and communication lines. In particular, they must respond to system failures (when primary location contents are lost), media failures (when volumes of secondary storage are lost), and network partitionings (when two or more subsets of system sites lose the ability to communicate with each other). (Communication line failures that do not partition the network can be masked by message re-routing). When any of these failures is announced by the

database operating system, a query optimization algorithm must adjust its execution strategies by excluding failed sites, or sites in foreign partitions from its list of available resources. Depending on the partitioning policy enforced by a given database management system, it may also be necessary to cease accessing certain data (sometimes all data) in a certain partition (sometimes in all but one partition)[BhLi]. After recovery from a failure, announced by the database operating system, the query optimization algorithm should adjust its execution strategies again.

(c). Logic queries

There are a number of proposals to optimize logical queries, i.e., queries are expressed in first-order logic as a collection of Horn clauses (see, for example, [HaLu, Ioan, JaCV, BMSu, SaZa, Nang, Ullma, Warr]). Our feeling is that while the ideas presented are reasonable, they have to be evaluated with realistic database systems. For example, the strategy in [Warr] suggests the rearrangement of subgoals so that the subgoal to be executed next is the one with least expected number of alternatives. It is assumed that all attributes are indexed and therefore all tuple accesses are equally costly. In realistic database environment, some attributes of certain relations have fast access paths, while others do not, as explained in the cost model of the replicate algorithm [YGCC, YGZT]. Thus, different tuples may be accessed with different costs.

Evaluation of a relation at a time [Ullma] instead of a tuple at a time is realistic in database environment. However, cost equations are not provided, which may prevent the construction of optimal or near-optimal strategies. [HaLu] introduced 3 rather general algorithms to process linear recursive queries. However, the bounding of variables are not considered. The analyses provided by [Ioan, Nang] are extremely useful but have applied to very restrictive class of queries. We believe that their analyses when applied to more general queries will yield very useful results.

Our approach is to modify existing proposals (for example[GaMN, HaLu, Ioan, NaHe, Nang, Warr, Ullma]), extend the research to distributed environments and test the performance in actual networks. Algorithms will be refined based on experimental observations.

4. Aim

The aim is to develop an algorithm that is capable of processing efficiently relational and logical queries in a general and dynamic environment. In order to accomplish this, it is essential that realistic experiments be performed on actual networks to gain understanding of the effect of different steps on performance.

REFERENCES

- [AphY] Apers P., Hevner A. and Yao S. B. OPTIMIZATION ALGORITHM FOR DISTRIBUTED QUERIES. IEEE Transactions on Software Engineering, 1983.
- [BeCh] Bernstein P. A. and Chiu D-M. W. USING SEMI-JOINS TO SOLVE RELATIONAL QUERIES. JACM, 1981, pp. 25-40.
- [BeGo] Bernstein P. A. and Goodman N. THE THEORY OF SEMI-JOIN. Technical Report, CCA, Nov. 1979.
- [BhLi] Bhargava, B. and Lilien, L., RELIABILITY ISSUES IN DISTRIBUTED

- DATABASE SYSTEMS. Technical Report 82-1, Dept. of Computer Science, Univ. of Pittsburgh, Pittsburgh, PA, Sept. 1982; to appear in Concurrency Control and Reliability in Distributed Systems, B. Bhargava, New York, Van Nostrand Reinhold.
- [BiDT] Bitton, D., DeWitt, D. and Turbyfil, C., "Benchmarking Database Systems: A Systematic Approach", VLDB, 1983.
- [BlLu] Black P. A. and Luk W. S. A NEW HEURISTEC FOR GENERATING SEMI-JOIN PROGRAMS FOR DISTRIBUTED QUERY PROCESSING. IEEE COMPSAC, 1982.
- [BrTY] Brill, D., Templeton, M. and Yu, C. "Distributed Query Processing Strategies in Mermaid : A Frontend to Data Management Systems", IEEE Data Engineering, 1985, pp. 211-218.
- [Brit] Britton Lee Inc. "IDM 500 Software Reference Manual", version 1.3, September 1981.
- [Call] Carey, M., Livny, M. and Lu, H., "Dynamic Task Allocation in Distributed Database Systems", IEEE Distributed Computing Systems, 1985.
- [CDFG] Chan A., Dayal U., Fox S., Goodman N., Ries D. and Skeen D. OVERVIEW OF AN ADA COMPATIBLE DISTRIBUTED DATABASE MANAGER. ACM SIGMOD 83, pp. 228-242.
- [CePe] Ceri, Stefano and Pelagatti, Giuseppe, "Distributed Databases Principles and Systems", McGraw-Hill Book Company, 1984.
- [Chan1] Chang J. M. A HEURISTIC APPROACH TO DISTRIBUTED QUERY PROCESSING. VLDB, 1982.
- [Chan2] Chang J. M. QUERY PROCESSING IN A FRAGMENTED DATA BASE ENVIRONMENT. Bell Lab., Technical report, 1982.
- [ChBH] Chiu, D. M., Bernstein, p., and Ho, Y. C., "Optimizing Chain Queries in a Distributed Database System", SIAM J. Comput., February 1984.
- [ChHo] Chiu D-M. W. and Ho Y. C. A METHOD FOR INTERPRETING TREE QUERIES INTO OPTIMAL SEMI-JOIN EXPRESSIONS. ACM SIGMOD, 1980, pp. 169-178.
- [ChHu] Chu, Wesley W. and Hurley, P., "Optimal Query Processing for Distributed Database Systems", IEEE Transactions on Computers, Vol c-31, No. 9, Sept. 1982, pp. 835-850.
- [ChLi1] Chen, A. L. P. and Li, V. O. K. "Deriving optimal semi-join programs for distributed query processing", Proc. IEEE INFOCOM, San Francisco, California, April 1984.
- [ChLi2] Chen, A. L. P. and Li, V. O. K. "Optimizing star queries in a distributed database system", VLDB, Singapore, August 1984.
- [ChLi3] Chen, A. L. P. and Li, V. O. K. "Improvement algorithms for semi-join query processing programs in distributed database systems", IEEE Transactions on Computers, Nov. 1984.
- [Date] Date, C. J. "An Introduction to Database Systems", Addison-Wesley, Reading, Mass., 1977.
- [Daya] Dayal, U., "Evaluating queries with quantifiers. A horticultural approach", ACM Symposium on Principle of Database Systems, 1983, pp. 125-136.
- [EpSW] Epstein R., Stonebreaker M. and Wong E. DISTRIBUTED QUERY PROCESSING IN RELATIONAL DATABASES SYSTEM. ACM SIGMOD 1978, pp. 169-180.
- [GaMN] Gallaire, H., Minker, J. and Nicolas J. M., "Logic and Databases: A Deductive Approach", ACM Computing Survey, 1984.
- [GoSh] Goodman N. and Shmueli O. TRANSFORMING CYCLIC SCHEMES INTO TREES. ACM SIGACT-SIGMOD Conference on Principles of Databases, 1982.
- [HaLu] Han, Jiawei and Lu, Hongjun, "Some Performance Results and Recursive Processing in Relational Database Systems", IEEE Data Engineering, 1986, pp. 533-539.
- [HeYa] Hevrer A. and Yao S. B. QUERY PROCESSING IN DISTRIBUTED DATABASE SYSTEMS. IEEE Transaction on Software Engineering, Vol. 5, No. 3, 1979,

- pp. 177-187.
- [Ioan] Ioannides, Y., "A Time Bound on the Materialization of Some recursively Defined Items", VLDB, 1985, pp. 219-226.
 - [JaCV] Jarke, M., Clifford, J. and Vassilion, Y. "An Optimizing Prolog Front-end to a Relational Query", ACM SIGMOD, 1984.
 - [JaKo] Jarke, M. and Koch, J., "Query Optimization in Database Systems", in ACM Computing Surveys, June 1984.
 - [KaYo] Kambayashi, Y. and Yoshikawa, M., "Query processing utilizing dependencies and horizontal decomposition", ACM-SIGMOD International Conference on Management of Data, San Jose, CA., May 1983. pp. 55-67.
 - [KaY] Kambayashi, Y., Yoshikawa, M. and Yagima, S., "Query processing for distributed databases using generalized semi-joins", ACM-SIGMOD International Conference on Management of Data, Orlando, Fla., June 1982. pp.151-160.
 - [KeYa] Kerschberg L. and Yao S. B. OPTIMAL DISTRIBUTED QUERY PROCESSING. Bell Lab., Holmdel, 1980.
 - [Kim] Kim, W., "Highly Available Systems for Database Applications", ACM Computing Survey, March 1984, pp. 71-98.
 - [LMHL] Lohman, G., Mohan, C., Hass, L., Lindsay, B., Selinger, P. and Wilms, P., "QUERY PROCESSING IN R*" IBM Research Report RJ4272, April, 1984.
 - [Luk2] Luk W. C. and Luk L. OPTIMIZING QUERY PROCESSING STRATEGIES IN A DISTRIBUTED DATABASE SYSTEM. Simon Fraser University, Burnaby, B. C. Canada.
 - [NaHe] Naqvi, S. A. and Henschen, L. T. "On compiling queries in recursive first-order databases", JACM 1984, pp.47-85.
 - [Nang] Nanghton, J., "Data Independence Recursion in Deductive Databases", ACM SIGACT-SIGMOD, 1986, pp. 267-279.
 - [Rein] Reiner D. (guest editor) IEEE Database Engineering, Special Issue On Query Processing, Sep., 1982.
 - [SDD1V1] Goodman N., Bernstein P. A., Wong E., Reeve .C. and Rothnie J. B. QUERY PROCESSING IN A SYSTEM FOR DISTRIBUTED DATABASES (SDD-1). Technical Report, CCA 1979.
 - [SDD1V2] Bernstein P. A., Goodman N., Wong E., Reeve C. and Rothnie J. B. QUERY PROCESSING IN SDD-1: A SYSTEM FOR DISTRIBUTED DATABASES. TODS, Vol. 6, No. 4, Dec. 1981, pp. 602-625.
 - [SaZa] Sacca, D. and Zaniolo, C., "On the Implementation of a Simple Class of Logic Queries for Databases", ACM SIGACT-SIGMOD, 1986, pp. 16-23.
 - [SeAd] Selinger, P., and Adiba, M. "Access Path Selection in Distributed Database Systems", Proceedings of The First International Conference on Distributed Data Bases, Aberdeen, 1980.
 - [TBHK] Templeton, M., Brill, D., Hwang, A., Kameny, I., and Lund, E. "An overview of the Mermaid system - A frontend to heterogeneous databases", IEEE Eascon83, Washington, Sept. 1983.
 - [TBCD] Templeton, M., Brill, D., Chen, A., Dao, S. and Lund, E. "Mermaid Experiences with Network Operations", IEEE International Conference on Data Engineering, Feb. 1986.
 - [Ullm] Ullman J. D. PRINCIPLES OF DATABASE SYSTEM. Computer Science Press, 2nd edition, 1982.
 - [Ullma] Ullman, J. D. "Implementation of Logic Query Languages for Databases", ACM SIGMOD, 1985 reprinted in ACM TODS Sept. 1985, pp. 289-321.
 - [VaGa] Valduriez, Patrick and Gardarin, Georges, "Join and Semi-join Algorithms for a Multiprocessor Database Machine", ACM Transactions on Database Systems, March 1984. pp. 133-161.
 - [Warr] Warren, D. "Efficient Processing of Interactive Relational Database

- Queries Expressed in Logic", VLDB, 1981, pp. 272-281.
- [WOHL] Williams et. al. R*: AN OVERVIEW OF THE ARCHITECTURE. Proc. 2nd International Conference on Databases, 1982.
- [WoKa] Wong, E. and Katz, R. H. "Distributing a database for parallelism", ACM SIGMOD, 1983, pp.23-29.
- [Wong] Wong E. RETRIEVING DISPERSED DATA FROM SDD-1: A SYSTEM FOR DISTRIBUTED DATABASES. Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, 1977.
- [Yaos] Yao, S. B. "Optimization of Query Evaluation Algorithms", ACM TODS, Vol 4, No. 2, June 1979, pp. 133-155.
- [YCTB] Yu C. T., Chang C. C., Templeton M., Brill D. and Lund E. ON THE DESIGN OF A DISTRIBUTED QUERY PROCESSING STRATEGY. ACM SIGMOD, 1983, pp. 30-39.
- [YCTBL] Yu, C. T., Chang, C. C., Templeton, M., Brill, D., and Lund, E. "Mermaid: An algorithm to process queries in a fragmented database environment", IEEE Transactions on Software Engineering, August, 1985, pp. 795-810.
- [YGBC] Yu, C. T., Guh, K. C., Brill, D., and Chen, A.L.P., "Partitioning Relation for Parallel Processing in Fast Local Networks", Dept. of EECS, Univ. of Illinois at Chicago, 1986.
- [YGCC] Yu C. T., Guh K. C., Chang C. C., Chen C. H., Templeton M. and Brill D. AN ALGORITHM TO PROCESS QUERIES IN A FAST DISTRIBUTED NETWORK. IEEE Real-Time Systems Symposium 1984, pp. 115-122.
- [YGZT] Yu, C. T., Guh, K. C., Zhang, W., Templeton, M., Brill, D., and Chen, A., "Algorithms to process Distributed Queries in Fast Local Networks", University of Illinois at Chicago, August 1985.
- [YLCC] Yu C. T., Lam K., Chang C. C. and Chang S. K. A PROMISING APPROACH TO DISTRIBUTED QUERY PROCESSING. Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, Feb. 1982, pp. 363-390.
- [YLGT] Yu, C. T., Lilien, L., Guh, K. C., Templeton, M., Brill, D. and Chen, A., "Adaptive Techniques for Distributed Query Optimization", International Conference on Data Engineering, Los Angeles, Feb. 1986.
- [YLGW] Yu, C. T., Lilien, L., Guh, K.C. and Wu, E. "A Graphical Interface for Learning in Distributed Query Processing", Dept. of EECS, Univ. of Illinois at Chicago, 1986.
- [YSLCa] Yu, C. T., Siu, M. K., Lam, K., and Chen, C. H. "Adaptive File Allocation in a Star-Network", IEEE COMPSAC 1983, pp.537-546. reprinted in IEEE TRANSACTIONS on Software Engineering, Sept. 1985.
- [YSLCb] Yu, C. T., Siu, M. K., Lam, K., and Chen, C. H. "File Allocation in Distributed Databases with Interaction Between Files" VLDB, Oct. 1983, pp.248-259.
- [YSLs] Yu, C. T., Suen, C., Lam, K., and Siu, M. K. "Adaptive Record Clustering", ACM Transaction on Data Base Systems, June 1985.
- [YuGC] Yu, C. T., Guh, K. C., and Chen, A.L.P., "An Integration for Two Distributed Query Processing Algorithms", Dept. of EECS, Univ. of Illinois at Chicago, 1985.
- [YuOz] Yu C. T. and Ozsoyoglu M. Z. AN ALGORITHM FOR TREE-QUERY MEMBERSHIP OF A DISTRIBUTED QUERY. IEEE COMPSAC, 1979, pp. 306-312.

A PARADIGM FOR CONCURRENCY CONTROL PERFORMANCE EVALUATION

A. A. Helal, A. K. Elmagarmid, and A. R. Hurson
Computer Engineering Program
Department of Electrical Engineering
The Pennsylvania State University
University Park, PA 16802
(814) 863-1047

ABSTRACT

Research in the area of concurrency control performance evaluation has extensively been addressed in the past few years. Unfortunately, conclusions arrived to by many researchers were inconsistent and at times contradictory. This inconsistency has arisen due to the different assumptions and performance models used. In order to eliminate this situation, a unified performance model is needed. In this paper, we propose a framework for concurrency control performance evaluation in single site databases. Areas of performance studies are classified, and a suggested performance model is specified.

1-INTRODUCTION

Despite the very large number of concurrency control algorithms in both single site and distributed databases, there does not exist any formal quantitative method for analyzing and comparing their performance. However, a number of performance studies were conducted in the last few years. Different assumptions and performance models were used in each of these studies, thereby leading to incomparable and in many cases contradictory results. Therefore, future performance studies should be based on a unified framework. In this paper, a framework for concurrency control performance evaluation in single site databases is proposed.

In section 2, an overview of database systems and performance studies is given, followed by the proposed performance evaluation model in section 3. Finally some concluding remarks are discussed in section 4.

2.0-OVERVIEW

2.1-DATABASES AND CONURRENCY CONTROL

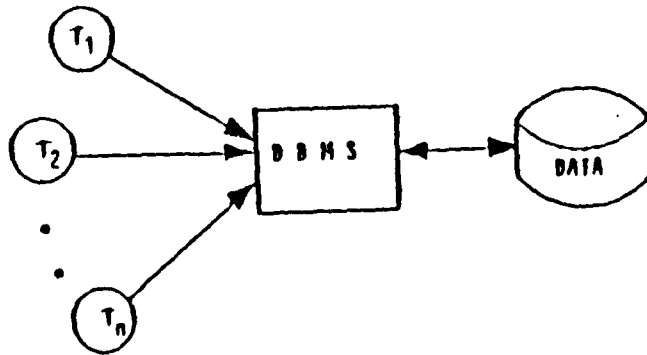


Figure 1 : Multiuser Database System

A single site database system (Figure 1) consists of three components: the transaction component, the database management system (DBMS) component, and the database component. Each component has a set of logical and/or physical parameters which constitute the overall system characteristics. User transactions interact with the DBMS by issuing read and write requests for database items stored in the database. The concurrent execution of transactions can result in data misuse (lost updates and dirty reads [Bern81]). Therefore, database systems include a subsystem called the concurrency controller to control concurrent accesses to the shared database.

A concurrency control algorithm (CCA), can be designed using one of three approaches: Locking [Mena78,Rose78], Timestamping [Reed78,Thom79,Bern81], or the Optimistic approach [Baye80,Kung81]. All previous approaches can be implemented to allow for multiple versions of a database [Reed78,Baye80,Care86].

2.2-OVERVIEW OF PERFORMANCE STUDIES

There are five areas to which performance studies should be directed. In the remainder of this section, each area is explained briefly.

System Throughput and Transaction Response Time

These are two indices of special importance. Minimizing the average transaction response time is of special interest to the user, whereas maximizing the system throughput is of special interest to the system manager. Most researchers have directed their efforts to this area.

Algorithm Efficiency and Asymptotic Behavior

These studies investigate the useless percentile of storage, CPU, I/O, and for the distributed case, intersite communications. This is called algorithm overhead, and it can be viewed as cost factors which affect the system throughput and the transaction response time. Many researchers directed their research to this area [Ries77,Ries79].

The Effect of Varying Transaction Component Parameters

The transaction component parameters affecting system throughput, transaction response time, and algorithm efficiency are listed below :

- 1- The number of transactions in the system (load or degree of multiprogramming).
- 2- Transaction size in terms of the number of read and write operations.
- 3- CPU resource requirements.
- 4- Read/Write mix (ratio of query to update).
- 5- Locality of requests.
- 6- Read/Write operation sequence (either interleaved or clustered into a read step followed by a write step).
- 7- The time between requests.
- 8- Transaction semantics (for formal definition of transaction semantics, see [Bhar84]).

Each of the above listed parameters has an impact on the overall system performance. Most researchers investigated 1, 2, and 3 above. In these studies, the degree of multiprogramming ranged from 5 to 500 transactions; small and large transaction classes were used to represent different transaction sizes. The effect of the fourth parameter (Read/Write mix) was studied by [Hela85,Care86] and there is no reported study addressing the effect of 5, 6, 7 and 8 above. It should be noted that transaction semantics do not directly affect the performance; however, CCAs can be designed to utilize transaction semantics to improve the performance.

The Effect of Varying the DBMS Component Parameters

The DBMS component parameters affecting system throughput, transaction response time, and algorithm efficiency are listed below :

- 1- The concurrency control algorithm.
- 2- The recovery technique.
- 3- Other modules such as : data access method, security module, and query optimization module.

The CCA has a great impact on performance and most of the research in this area has been focused on : two-phase locking, basic timestamp ordering, serial validation and multiversion algorithms. Recovery techniques affect the way CCAs are designed and consequently the overall system performance. Two-phase commit is the most widely used technique in single site databases. Although the effect of the various recovery techniques can be studied separately, the effect of each technique on the CCA should be investigated. The effect of the two-phase commit on the dynamic two-phase locking and the serial validation algorithm was studied by [Hela85].

The Effect of Varying the Database Component Parameters

The database component parameters affecting system throughput, transaction response time, and algorithm efficiency are listed below :

- 1- The data model (relational, hierarchical, network, etc ...)
- 2- The physical database size (given in terms of physical units called items)
- 3- The granularity (a granule may contain more than one item)

Like transaction semantics, the data model has no direct effect on the performance, but a CCA can be designed to utilize a specific data model for better performance. Granularity is a critical parameter which dramatically affects the overall system performance, especially when the system is loaded. Ries and Stonebraker [Ries77, Ries79] have investigated the effect of "locking" granularity on the performance of DBMS. However, granularity should also be studied for non-locking algorithms.

3-THE PROPOSED PERFORMANCE EVALUATION MODEL

In this section, a performance model to be used in evaluating the performance of concurrency control algorithms is given. The model consists of four components : the transaction workload description, the system structure and parameters, the concurrency control algorithm, and the performance indices. In the remainder of this section, these components are explained first and then a set of model assumptions are stated.

(a) Transaction Workload Description

Transaction workload can be specified using the following parameters :

1- The number of transactions in the system (degree of multiprogramming) or in case of open system modeling, the average arrival rate and the distribution of the arrival process (usually assumed to be poisson).

2- Transaction size which is the number of the read and write operations in a transaction. This can either be deterministic (fixed size with the possibility of multiple transaction classes) or probabilistic (usually uniformly distributed over [min_size...max_size]).

3- The amount of CPU time required by a transaction. This can either be neglected or assumed to be exponentially distributed with some mean.

4- The Read/Write mix which can either be deterministic (e.g. : 80% read and 20% write) or probabilistic (e.g. : prob {next request is read} > 0.2). It is possible to have different transaction classes that range from pure query to heavy update.

5- The Read/Write operations sequence. It can be either interleaved or clustered.

6- The distribution of the time between access requests. This is not known to have a certain distribution and can be assumed uniform.

7- Transaction semantics. They can be described by redefining the read and write operations as pairs (P, Operation) where P is some predicate that when evaluated True, the R/W operation would be performed. We give no specification for the predicate P.

(b) The System Structure

The system structure is depicted in Figure 2. This structure can be used in modeling both open and closed systems. When a new transaction arrives, its workspace is initialized and it is put on the concurrency controller queue ,CCQ. The concurrency controller, CC then serves the transaction by : (i) allowing it to the database, (ii) blocking it in the block queue, BQ, (iii) aborting it, or (iv) committing it. In the case of a non-blocking CCA, the BQ is not used. If the transaction must wait for an unavailable resource, it is put on the BQ. When the resource is available, it is dequeued and is put back on the CCQ. Once allowed into the system, a transaction with a read request is placed on the query queue, QQ. However, in case of a write request, no disk write operation takes place and the new value is updated only in its workspace. When a transaction finishes, the CC puts it on the update queue, UQ so that all its updates may be

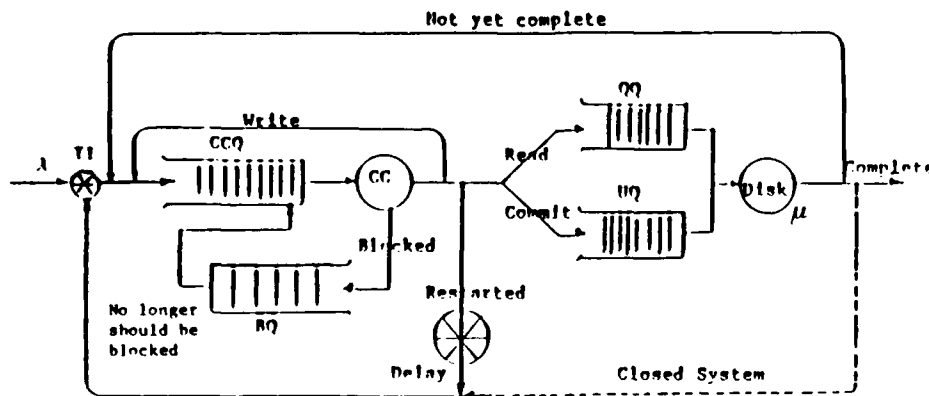


Figure 2 : System Structure

committed. The UQ has higher priority than the CC. If a transaction is aborted, the CC resets or modifies its workspace, possibly delays it, and then resubmits it to the system through the transaction initiator, TI. When a new transaction arrives at the TI, the later submits a small setup transaction with zero or one read operation. This setup transaction is guaranteed to commit with no restarts and it represents a setup cost for newly arriving or restarted transactions. When the setup transaction commits, the original transaction is considered by the CC.

(c) The Concurrency Control Algorithm

The concurrency control algorithm is specified by a piece of actual code. Since this code is sharable by different transactions, it is considered as a critical section and is modeled by a single server queuing model with FIFO discipline and with service time found by one of two methods. First, a counter can be used to count the number of actually executed statements while calling this code. This exact count gives an exact service time for the transaction being served. This exact service time can then be used as a predicted service time for the next transaction to be served. Second, using the first method, simulation can be conducted to compute the average service rate which can then be used in subsequent experiments.

(d) The Performance Indices

Following are some general indices that should be measured in any performance study.

- 1- Average transaction response time.
- 2- System throughput.
- 3- Average degree of concurrency (DC)
- 4- Conflict rate.
- 5- Maximum number of times a transaction was restarted.

6- Useless I/O and CPU percentile.

There are also certain indices applicable only to specific CCAs. An example is the deadlock rate in the two-phase locking algorithm. To define the degree of concurrency, DC, let T_n be the duration of time through which the number of active transactions in the system is n . The degree of concurrency is then given by :

$$DC = 1/(NT_s) * \sum_{n=1}^N nT_n$$

Where T_s is the system time at which the DC is measured and N is the number of transactions in the system. Clearly, $0 < DC < 1$.

MODEL ASSUMPTIONS

1- Two-phase commit is incorporated with the CCA used. Thus a write operation is treated as a non-disk operation and is committed on transaction completion. Two-phase commit begins at transaction completion by issuing $1 + |\text{writeset}|$ disk operations (one in the first phase and $|\text{writeset}|$ in the second phase).

2- Resources are finite : the physical database is stored in one disk unit with some service rate. The case of more than one disk unit can equivalently be studied by one disk unit with a higher service rate. Also, there is only one CPU with some speed in the system.

3- Elements of the readset and the writeset of a transaction are distinct.

4- A transaction workspace consists of two parts : the transaction definition and the transaction data section. When a transaction is restarted, only its data section is reset.

4-CONCLUSION

In this paper, we have classified the research studies of concurrency control performance evaluation into five major areas, three of which reflect the effect of the database environment on the performance. Also, we proposed a framework for performing these studies using a unified performance model.

REFERENCES

- [Baye80] Bayer, R., Heller, H., and Reiser, A., Parallelsim and recovery in database systems, ACM trans. on DBS 5(2), June, 1980, 139-156.
- [Bern81] Bernstein, P., and Goodman, N., Concurrency control in distributed database systems. ACM computing Survey, 13(2), June 1981, 185-221.
- [Bhar84] Bhargava, B., Concurrency control and reliability in distributed database management systems, Handbook of software engineering, North Holland, 84, 331-358.
- [Care86] Carey, M., and Muhana, W., The performance of multi-version concurrency control algorithms. (To appear in ACM TOCS).
- [Hela85] Helal, A., Performance analysis of concurrency control algorithms in database systems. M.Sc. thesis, Computer Science Department, Alexandria University, Egypt, July, 1985.
- [Kung81] Kung, H., and Robinson, J., On optimistic methods for concurrency control. ACM trans. on DBS, 6(2) June, 81, 213-226.
- [Mena78] Menasce, D., and Muntz, R., Locking and deadlock detection in distributed databases. Proc. of the 3rd Berkeley workshop on distributed data management and computer networks. August 1978.
- [Reed78] Reed, D., Naming and synchronization in a decentralized computer system, Ph.D. thesis, Dept. of EE and Computer Science, MIT, 1978.
- [Ries77] Ries, D., and Stonebraker, M., Effects of locking granularity in a database management system. ACM trans. on DBS, 2(3), Sep. 1977, 233-246.
- [Ries79] Ries, D., and Stonebraker, M., Locking granularity revisited, ACM trans. on DBS, 4(2), June 1979, 210-227.
- [Rose78] Rosenkrantz, D., Stearns, R., and Lewis, P., System level concurrency control for distributed database systems. ACM trans. on DBS, 3(2), June 1978, 178-198.
- [Thom79] Thomas, R., A majority consensus Approach to concurrency control for multiple copy databases. ACM trans. on DBS, 4(2), June, 1979, 180-209.

Concurrency Control and Reliability in Replicated Database Systems

Mukesh Singhal

Dept. of Computer and Information Science
The Ohio State University
2036 Neil Avenue Mall
Columbus, OH 43210

Abstract

Distributed database systems are of utmost importance to military exercises because such applications are inherently distributed and require severe performance constraints such as fast response time and continued operation in face of catastrophic failures. In this talk, we examine several issues in the design of a high performance distributed database system. In particular, we discuss the problem of *concurrency control* which deals with controlling the concurrent access of a database by simultaneously running transactions such that the correctness of the database is maintained, and the problem of *reliability* which deals with the resiliency to different kinds failures. We examine the state of the art of these design issues and discuss future directions for distributed database systems.

1. Introduction

A database is a collection of related data objects, which is shared by several users with potentially diverse interests. The data objects of a database must usually satisfy certain semantic relationships, referred to as the *consistency assertions* of the database [22]. A database is *consistent* if values of its data objects satisfy all of its consistency assertions.

The falling cost of hardware and advances in communication technology over the last decade have made distributed computing viable and have triggered interest in *distributed database systems* (e.g., [7, 45, 48]). In such systems, data objects are spread over a collection of computers which are connected via some communication network. A distributed database system offers several advantages over a single-site database system [1], such as data and program sharing, higher system throughput, higher system availability, load sharing, and easy expandability.

In a *replicated database system*, all the data objects of a database are duplicated at every site. Replication of data objects offers several attractive features: enhanced reliability, improved responsiveness, no directory management, and easier load balancing. Enhanced reliability results because a site crash or network partitioning does not prevent access to some data objects and, in general, does not stop transaction processing under these failures. Because of the enhanced reliability, replicated database systems are highly desirable in hostile environment where site crashes or network partitionings are inevitable and continued

operation under these failures is very crucial. These features have led to several commercial efforts in the direction of replicated database systems, e.g., SDD-1 [6] and distributed INGRES [45].

However, a replicated database system must guarantee that all the copies of a data object agree with each other. This additional constraint is reflected as a consistency assertion. As a consequence, in a replicated database system, the concept of consistency has two parts [46], internal consistency and mutual consistency. *Internal consistency* deals with the relationships of data objects within a database copy. *Mutual consistency* requires that all database copies must be identical.

A user interacts with a database by performing read and write actions on the data objects. The actions of a user are normally grouped together (as a program) to form a single logical unit of interaction which is referred to as a *transaction*. To enhance the efficiency, the actions from several transactions may be executed in an interleaved manner; however, for correctness, the system must behave as if each transaction is processed atomically, i.e., as an indivisible unit. There are two kinds of transaction atomicities: concurrency atomicity, and failure atomicity. *Concurrency atomicity* enforces that concurrent execution of transactions must behave as if transactions are executed serially. *Failure atomicity* enforces all or none property of the execution of a transaction in presence of failures.

There are several issues in distributed database systems which must be efficiently resolved[37]. For examples, concurrency control [5, 22], deadlocks[19, 31]; directory management[16]; optimal program and data allocation[34]; distributed transaction processing[17, 30]; atomic commit[28, 32]; and crash recovery[28, 17]. In this talk, we limit our discussion to concurrency control and reliability aspects of a distributed database.

2. Concurrency Control

Typically, in a database system, several users concurrently access the database by executing transactions. For efficiency reason, the system executes the actions from several transactions in an interleaved manner. Since the actions of concurrently running transactions may access the same data objects, if interleaving of the actions is not controlled in some orderly way, several anomalous situations may arise. For examples, lost updates, incorrect retrieval, and inconsistent update. As a result, some transactions may see an inconsistent state of a database, and a database may terminate in an inconsistent state. This fundamental problem is referred to as *concurrency control*. In a database system, this problem is handled by a concurrency control algorithm which controls the relative order (or interleaving) of conflicting¹ actions, so that the consistency of a database is preserved.

Most concurrency control algorithms are based on locking or timestamps. In *locking* based algorithms, a transaction must lock a data object before accessing it [25]. A transaction can lock a data object if it is not already locked by some other transaction. In *timestamp* based algorithms, every site maintains a logical clock which is incremented by one when a transaction arrives at that site and updated whenever the site receives a message with higher clock value (every message contains current clock value of its sender site). Each transaction is assigned a unique timestamp and conflicting actions are interleaved in order of

¹ Two actions conflict if they operate on the same data object, and at least one of them is a write action.

the timestamp of their transactions.

2.1. Current Status

Several concurrency control algorithms have been proposed for replicated database systems. Depending upon whether synchronization is performed before, or after an update makes an access to the data objects, these algorithms can be divided into two classes: *pessimistic*, and *optimistic* [18]. Algorithms that perform synchronization before accessing data objects are referred to as *pessimistic* algorithms. Algorithms that perform synchronization after accessing data objects are referred to as *optimistic* algorithms.

Pessimistic Algorithms

In pessimistic algorithms, a site executes an update after carrying out some intersite communication for synchronization. Depending upon the way intersite communication and synchronization are performed, several pessimistic algorithms have been proposed in the literature, e.g., [2, 6, 8, 21, 25, 33, 35, 39, 45].

Optimistic Algorithms

In optimistic algorithms, an update is executed concurrently with other updates without performing any synchronization. However, before its computed values are written into the database, some intersite communication is carried out to determine if the update has conflicted with any concurrently executing updates. In case of conflicts, the lower priority update is aborted, else its computed values are written into the database. Depending upon the way intersite communication and check for conflict are carried out, several optimistic algorithms have been proposed in the literature, e.g., [15, 18, 27, 35, 46].

Performance

The performance of concurrency control algorithms for replicated database systems has been studied using simulation techniques (e.g., [27, 15, 25, 42]), as well as analytic techniques (e.g., [26, 42]). The performance of a concurrency control algorithm is usually measured by *update response time* which is defined as the time interval between the instant when a user submitted an update at a site and the instant when the update is completely executed at that site, and *system throughput* which is the number of transactions completed per unit of time in a system.

In pessimistic algorithms, a transaction locks out (i.e., blocks) all the concurrent conflicting transactions. If conflicts among transactions are frequent, then the performance of these algorithms is limited due to large blocking delays. In optimistic algorithms, a conflict causes a abort and restart of a transaction. Each abort and restart increases the response time of the update and wastes computing and communication resources. The performance of optimistic algorithms is limited because frequent conflicts among updates induce repeated aborts and restarts.

2.2. Future Directions

High Performance Algorithms

Since concurrency control is a fundamental problem in database systems, a high performance database system calls for a high performance (i.e., short response time and/or higher throughput) concurrency control algorithms. The performance of the most of the current concurrency control algorithms is limited by the blocking delays due to conflicts. Therefore, development of high performance concurrency control algorithms requires reducing the blocking delays due to conflicts. Since conflicts among transactions are inherent to a system, we require radically different techniques for concurrency control (where the performance is not limited by blocking delays due to conflicts).

We have developed a new technique for concurrency control in replicated database systems [41], where the blocking delays due to conflict depend upon the speeds of disk (I/O device) and cpu rather than the speed of communication network (as in existing algorithms). Since disk and cpu are usually much faster than the communication medium, the new approach exhibits a substantial improvement in the performance. We conducted a performance study [42], which shows that the new technique has much better update response time and throughput characteristics as compared to existing algorithms (unless disk is very slow and/or message propagation delay is small).

An interesting finding of the study is that the maximum throughput of the centralized locking algorithm [25], and Milenkovic's pessimistic algorithm [35] (for that matter of all existing algorithms) is limited because updates inflict large blocking delays due to conflicts. On the contrary, the maximum throughput of the new technique is limited by the disk throughput. Since disk speed can be readily controlled by using a disk of higher speed or by using more disks, the performance of this technique can be easily controlled. Whereas, the performance of the existing algorithms is limited by large blocking delays due to conflicts among updates. Since conflicts among updates are inherent to a system and are difficult to control, the performance of these algorithms is difficult to control.

Performance Analysis

Although many concurrency control algorithms have been proposed, little progress has been made in the direction of their performance analysis. Queueing models of concurrency control algorithms for database systems have two distinct features, viz. *multiple resource possession* and *blocking* [14], which are absent in the queueing models of conventional systems. Because of these features, queueing models of concurrency control algorithms are not amenable to the product form solutions [1], and which makes the task of the performance analysis of concurrency control algorithms very difficult. Since performance analysis of concurrency control algorithms for distributed database systems is very important, there is a desperate need to develop new tools and techniques to cope with the complexities of concurrency control algorithms. Past methods for performance analysis of concurrency control algorithms [26, 40] have dealt with this complexity by relying upon the so called *approximation techniques* [14].

3. Reliability Issues

Reliability is highly desirable in military exercises where continued operation under catastrophic failures is very crucial. Reliability management encompasses several issues such as atomic commit [32, 44], site recovery [3, 47], network partitioning [11, 20], and fault tolerance [9].

Atomic Commit

In a distributed database system, a transaction is committed atomically if it is processed at all sites or at none. Designing an atomic-commit protocol which is resilient to arbitrary site failures and network partitionings is an extremely difficult task. The two-phase commit protocol [28, 32] achieves the atomic commit in the environment where sites can unilaterally abort a transaction. However, it is vulnerable to site failures as a site failure may block a transaction. In [29], the two-phase protocol is extended to four-phase protocol where in case of a site failure, a backup site takes over. In [44, 43], nonblocking protocols have been examined where a site never blocks a transaction because of some failures.

Crash Recovery

When a site recovers from a failure, the state of its database must be restored to a consistent state. The techniques used to restore the state of a database to a consistent state are called recovery techniques. Most recovery techniques retain some redundant information about the database or history of transaction execution, and recover a site by restoring its database to a previous consistent state. Some examples of redundant information are *audit-trail or log* [10, 28], *checkpoint* [23], *differential files* [38]. The recovery in SDD-1 is based on the notion of *spoolers* [29], where when a site fails, all messages directed to it are buffered at its spooler sites. When a site recovers, it unspools all messages buffered for it from its spooler sites. In DDM [12, 13], data availability is improved by using *incremental recovery* [3], and intersite data transfer in case of failures is reduced by using log-based recovery mechanism.

Network Partitioning

If transaction processing continues in presence of a network partitioning, the state of the database of each partition may diverge with time and the consistency of the database may be violated. Therefore, when a database system recovers from a partitioning, the state of the database in each partition must be reconciled to a common value. Every site keeps log or audit trail, and when a database recovers a partitioning, databases in different partitions are reconciled by merging the journals of different partitions to get a common journal and applying it to all the sites of the partitions being merged. Techniques for consistently integrating partitions of a replicated database system are described in [11, 20].

Fault Tolerance

A system is fault tolerant if in presence of certain faults, it automatically detects, isolates, and recovers from them to avoid a failure. SEQUOIA[9] is a tightly-coupled fault-tolerant multi-processor transaction processing system which uses hardware approach to fault-tolerance. It employs three kinds of mechanisms: (1) *error detecting codes*—while data is stored in a memory or while being transferred between different modules, they are

protected by error-detecting codes; (2) *duplication and comparison* - some functions such as address generation and cache management are duplicated in hardware. These components perform operations independently and cross check their results with those produced by their duplicates; and (3) *protocol monitoring* - prevents starvation due to malfunctioning of a components by detecting violations in the sequence and timing of inter-element communication.

Future Directions

Most existing reliability mechanisms block transaction processing or perform major reconfiguration in case of a failure which may degrade the performance in the presence of failures. This may be unacceptable in hostile environments because high performance (availability) is most desired when some attack is underway. One approach to handle this problem is to sacrifice correctness (consistency) of a system to obtain high availability and fast response. Some work in this direction is reported in [24, 36]. More work still need be done in the direction of formalization of the tradeoffs between the correctness and the performance, and new techniques need be developed to achieve these tradeoffs.

References

1. A. LYNCH, "Distributed Processing Solves Main-frame Problems," *Data Communications*, pp. 17-22 (December 1976).
2. ALSBERG, P. A. AND DAY, J. D., "A Principle for Resilient Sharing of Distributed Resources," *Proc. 2nd International Conf. on software Engineering*, pp. 562-570 (Oct. 1976).
3. ATTAR, R., BERNSTEIN, P. A., AND GOODMAN, N., "Site Initialization, Recovery, and Backup in a Distributed Database System," *IEEE Trans. on Software Engineering*, pp. 645-650 (Nov. 1984).
4. BASKET, F., CHANDY, K. M., MUNTZ, R. R., AND PALACIOS, F. G., "Open, Closed, and Mixed Networks of Queues with Different Classes of Customers," *Journal of ACM*, pp. 248-260 (April 1975).
5. BERNSTEIN, P. AND GOODMAN, N., "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys*, pp. 185-222 (June 1981).
6. BERNSTEIN, P. A., ROTHANIE, JR., J. B., GOODMAN, N., AND PAPADIMITRIOU, C. A., "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The Fully Redundant Case)," *IEEE Trans. on Software Engineering*, pp. 154-168 (May 1978).
7. BERNSTEIN, P. A., SHIPMAN, D. W., AND ROTHANIE, J. B. JR., "Concurrency Control in a System for Distributed Databases (SDD-1)," *ACM Trans. on Database Systems*, pp. 18-51 (March 1980).
8. BERNSTEIN, P. A. AND GOODMAN, N., "An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases," *ACM Trans. on Database Systems*, pp. 596-615 (Dec. 1984).
9. BERNSTEIN, P. A., "SEQUOIA: A Fault-Tolerant Tightly-Coupled Computer for Transaction Processing," *Technical Report TR-85-03, Wang Institute of Graduate Studies*, (May 1985).

10. BJORK, L. A., "Generalized Audit Trail Requirements and Concepts for Database Applications," *IBM Systems Journal*, pp. 229-245 (March 1975).
11. BLAUSTEIN, B.T., GARCIA-MOLINA, H., RIES, D.R., CHILENSKAS, R.M., AND KAUFMAN, C.W., "Maintaining Replicated Databases Even in Presence of Network Partitions," *EASCON*, pp. 353-360 (1983).
12. CHAN, A. AND SKEEN, D., "The Reliability Subsystem of a Distributed Database Manager," *Technical Report CCA-85-02, Computer Corporation of America, Cambridge, MA 02142*, ().
13. CHAN, A., DAYAL, U., FOX, S., GOODMAN, N., SKEEN, D., AND RIES, D., "DDM: An Ada Compatible Distributed Database Manager," *IEEE COMPCON Digests of Papers*, (1983).
14. CHANDY, K. M. AND SAUER, C. H., "Approximate Methods for Analysis of Queueing Network Models of Computer Systems," *Computing Surveys*, pp. 263-280 (Sept. 1978).
15. CHENG, W. K. AND BELFORD, G. G., "Update Synchronization in Distributed Databases," *Proc. of 6th Int. Conf. on Very Large Databases*, , pp. 301-308 (Oct. 1980).
16. CHU, W. W. AND NAHOURAH, E. E., "File Directory Design Considerations for Distributed Databases," *Intl. Conf. on Very Large Databases*, pp. 543-545 (1975).
17. CHU, W. W. AND HURLEY, P., "A Model for Optimal Processing for Distributed Databases," *Proc. of 18th IEEE COMPCON*, pp. 116-122 (Spring 1979).
18. CHU, W. W. AND HELLERSTEIN, J., "The Exclusive-Writer Approach to Updating Replicated Files in Distributed Processing Systems," *IEEE Trans. on Computers*, pp. 489-500 (June 1985).
19. COFFMAN, E. G., ELPHICK, M. J., AND SHOSHANI, A., "System Deadlocks," *ACM Computing Surveys*, pp. 66-78 (June 1971).
20. DAVIDSON, S.B., "Optimism and Concurrency in Partitioned Distributed Database Systems," *ACM Trans. on Database Systems*, pp. 456-481 (Sept. 1984).
21. ELLIS, C. A., "Consistency and Correctness of Duplicate Database Systems," *6th Symposium on Operating Systems Principles*, pp. 67-84 (1977).
22. ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L., "The Notion of Consistency and Predicate Locks in a Database System," *Communications of ACM*, pp. 624-633 (Nov. 1976).
23. FISCHER, M. J., GRIFFETH, N. D., AND LYNCH, N. A., "Global States of a Distributed System," *IEEE Trans. on Software Engineering*, pp. 198-202 (May 1982).
24. FISCHER, M. J. AND MICHAEL, A., "Sacrificing Serializability to Attain High Availability in an Unreliable Network," *Proc. of the First ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, (1982).
25. GARCIA-MOLINA, H., "Performance Comparison of Two Update Algorithms For Distributed Databases," *Proc. of 3rd Berkeley Workshop on Distributed Data Management and Computer Networks*, pp. 108-119 (Aug. 1978).
26. GARCIA-MOLINA, H., "Performance of the Update Algorithms for Replicated Data in a Distributed Database," Ph.D. Dissertation, Computer Science Dept., Stanford University (June 1979).
27. GARDARIN, G. AND CHU, W. W., "A Distributed Control Algorithm for Reliably and Consistently Updating Replicated Databases," *IEEE Trans. on Computers*, pp. 1060-1068 (Dec. 1980).
28. GRAY, J. N., "Notes on Database Operating Systems," pp. 393-481 in *Operating Systems: An Advance Course*, Springer-Verlag, N.Y. (1978).
29. HAMMER, M. AND SHIPMAN, D., "Reliability Mechanism for SDD-1: A System for Distributed Databases," *ACM Trans. on database Systems*, pp. 431-466 (December 1980).

30. HEVNER, A. R., "The Optimization of Query Processing on Distributed Database Systems," *Ph.D. Dissertation*, Dept. of Computer Science, Purdue University, West Lafayette, IN, (Dec. 1979).
31. ISLOOR, S. S. AND MARSLAND, T. A., "The Deadlock Problem : An Overview," *Computer Magazine*, pp. 58-77 (Sept. 1980).
32. LAMPSON, B. AND STURGIS, H., *Crash Recovery in a Distributed Data Storage System*, Tech. Report, Computer Science Lab., XEROX PARC, Palo Alto, CA (1976).
33. LELANN, G., "Algorithms for Distributed Data Sharing Systems Which Use Tickets," *Proc. of 3rd Berkeley Workshop on Distributed Data Management and Computer Networks*, pp. 259-272 (Aug. 1978).
34. MAHMOUD, S. AND RIORDON, J. S., "Optimal Allocation of Resources in Distributed Information Networks," *ACM Trans. on Database Systems*, pp. 66-78 (March 1976).
35. MILENKOVIC, M., "Synchronization of Concurrent Updates in Redundant Distributed Databases," *Distributed Data Bases*, pp. 49-65 North-Holland Publishing Co., (1980).
36. PARKER, D. S. AND RAMAS, R. A., "A Distributed File System Architecture Supporting High Availability," *Proc. of the Intl. Conference on VLDB*, (1982).
37. ROTHANIE, J., "A Survey of Research and Development in Distributed Database Management," *Proc. of the 3rd Conference on Very Large Databases, Tokyo*, (1977).
38. SEVERANCE, D. G. AND LOHMAN, G., "Differential Files: Their Application to the Maintenance of Large Databases," *ACM Trans. on Database Systems*, pp. 256-267 (Sept. 1976).
39. SINGHAL, MUKESH AND AGRAWALA, A. K., "A Concurrency Control Algorithm and its Performance for Replicated Database Systems," *To appear in Proc. of the 6th International Conference on Distributed Computing Systems, Cambridge, Massachusetts*, (May 19-23, 1986).
40. SINGHAL, MUKESH AND AGRAWALA, A. K., "Performance Analysis of an Algorithm for Concurrency Control in Replicated Database Systems," *To appear in Proc. of the Performance 86 and ACM-SIGMETRICS 86, Joint Conference on Computer Performance Modeling, Measurement, and Evaluation, Raleigh, North Carolina*, (May 28-30, 1986).
41. SINGHAL, M., "Update Transport: A New Approach to Update Synchronization in Replicated Database Systems," *Submitted to IEEE Trans. on Software Engineering*, ().
42. SINGHAL, M., "Concurrency Control Algorithms and Their Performance in Replicated Database Systems," *Ph.D. dissertation*, Dept. of Computer Science, University of Maryland, College Park, (February, 1986).
43. SKEEN, D., "Nonblocking Commit Protocols," *SIGMOD Intl. Conference on Management of Data*, (1981).
44. SKEEN, D. AND STONEBRAKER, M., "A Formal Model of Crash Recovery in a Distributed System," *IEEE Trans. on Software Engineering*, pp. 219-228 (May 1983).
45. STONEBRAKER, M., "Concurrency Control and Consistency of Multiple Copies in Distributed INGRES," *IEEE Trans. on Software Engineering*, pp. 188-194 (May 1979).
46. THOMAS, R. H., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Trans. of Database Systems*, pp. 180-209 (June 1979).
47. VERHOFSTAD, J.S.M., "Recovery Techniques for Database Systems," *ACM Computing Surveys*, pp. 167-195 (June 1978).
48. WILLIAMS, R., DANIELS, D., HAAS, L., LOPIS, G., LINDSAY, B., NG, P., OBERMARCK, R., SELINGER, P., WALKER, A., WILMS, P., AND YOST, R., "R*: An Overview of the Architecture," IBM Research Report RJ 3325, San Jose, CA (December 1981).

Session 13: MIMD Parallelism and Support

Chairperson: R. R. Oldehoeft
Colorado State University

Predicate Analysis for Parallel Program Generation

by

Boleslaw K. Szymanski
Computer Science Department
Rensselaer Polytechnic Institute
Troy, NY 12180

ABSTRACT

New software development tools proposed for supercomputers are based on assertive programming, where a program is expressed as a set of assertions about its properties and not as a sequence of steps leading to the solution. Solving procedures are automatically generated from assertive description.

Assertive programming for scientific parallel processing is supported by equational languages in which assertions are restricted to conditional equations. Such languages are naturally suited for mathematically oriented computations. Equational languages also proved to be an effective tool for describing general computational tasks. Descriptions of many parallel algorithms are greatly simplified when written in that form.

The MODEL equational language translator and its three cooperating components: compiler, configurator, and timer are discussed in this paper. The focus is on the application of predicate analysis and automatic theorem proving in area of arithmetic inequalities to checking equational programs and optimization of generated parallel code.

1. Introduction

Software development for parallel and vector computers is intrinsically more difficult than for sequential machines. In addition to the usual challenges, one must consider communications overhead and synchronization problems. Often performance evaluation is necessary to balance the computational load assigned to processors of a parallel computer. However, time performance can be evaluated only after the software has been developed and, therefore, leads to costly post-development tuning. Specialized parallel computing hardware often requires software to be developed in dialects of standard languages. In particular, varying communication primitives of different parallel computers lead to major software incompatibility problems. A focus on software efficiency additionally contributes to the high costs of software development. Distributed environments today and massive parallel computers of the future force the research community to seek novel approaches to parallel programming.

Many new approaches are based on assertive programming. In this paradigm, a computational problem is expressed as a set of assertions about its properties and not as a sequence of steps leading to the solution. Solution procedures are automatically generated from the assertive description. Users are not involved in the implementation, whose efficiency and correctness are assured by the underlying language processor.

Depending on the type of assertions used as a basis for notation, different languages for assertive programming have been proposed. Perhaps the best known is Prolog, in which assertions are expressed as Horn clauses. Automatic inference of new facts from the given rules and known facts makes it an ideal programming tool for artificial intelligence and expert systems. However, in an important area of scientific computing, particularly amenable to parallel processing due to regularity of problems and vast required computations, its usefulness is limited because of difficulty of expressing numerical algorithms in Prolog.

Another important paradigm of assertive programming is based on equational languages, where assertions are expressed as mathematical equations (cf. Ashcroft and Wadge (1977), Hoffman and O'Donnell (1982), and Prywes and Szymanski (1985)). Such languages are naturally suited for mathematically oriented computations. They are especially convenient for solving systems of linear equations that may arise directly (as, e.g., in econometric modeling) or as a result of a discrete approximation of a system of differential equations. Equational languages have also been proven to be an effective tool for describing general computational tasks (cf. Baron et al. (1985)).

An equational language can simplify programming, but it puts additional burden on a compiler. For example, flow of control has to be entirely defined by the compiler. Each variable can assume only a single value yielded by the defining equation. Redefinitions of a variable value, in the sense of procedural programming, have to be represented by a vector of values, i.e., a new variable with one dimension more than redefined one. Such additional dimension, if not recognized automatically, leads to unnecessary copying of redefined structures and low efficiency of generated object code (cf. Szymanski and Prywes (1986)). Therefore compiler of an equational language has to perform global and comprehensive analysis of specifications. Such overhead can be avoided if user is allowed to define flow of control, like in procedural languages for sequential processing. For parallel processing however, user's defined control is nearly optimal because of the complexity of the task. Therefore, a comprehensive analysis has to be performed anyway. A procedural program in this case is overspecified, increasing the chances of the user to introduce errors and limiting compiler in optimization because of unnecessary dependencies imposed by the flow of control (particularly nesting and scope of loops in iterative, numerical computations). Clearly, a nonprocedural, definitional specification is more convenient for parallel processing than procedural one.

Compilers for equational languages have to include tools for static analysis and optimization of generated code. Required analytic power is beyond the scope of traditional compiler optimization and flow of control analysis. Moreover the object code is generated in a high-level language and therefore it is exposed to these methods anyway. Therefore in designing equational language compilers, there is a need to concentrate on problems specific to nonprocedural specifications. Among the most important are: finding the optimal sequence of object program events, reducing the dimensionality of data structures, recognizing circular dependencies to be solved as simultaneous equations, and assessing consistency of specifications. Many of these problems were already addressed in

the MODEL compiler (cf. Szymanski and Prywes (1986)). A new approach to predicate analysis is discussed in this paper.

The paper focuses on practical application of theorem proving techniques in designing compilers for future software systems. It also describes how the use of an equational language and translator can simplify parallel programming. The fundamental notion underlying this approach is that prospective users have the knowledge and expertise in the application domain and not in computer programming. Users specify a *problem* progressively, by providing its mathematical model with little regard for how to implement its *solution*. This is done through composing equations that express the underlying rules, relations and concepts of the problem environment and not the solution. The user composes the specification cooperatively with an automatic system which performs two tasks. First, it verifies completeness, non-ambiguity and consistency of the user's input and solicits corrections and amplifications until it is satisfied with the comprehensiveness of the specification. Second, it automatically designs, programs, and schedules the corresponding parallel computation. The automatic system also provides timing information and reports its design decisions.

2. Description of the MODEL Equational Language and System

The discussion in the paper is based on our experience with the MODEL equational language and its translator (Tseng et al. (1986)). The MODEL language provides the user with a static view of computations. It shields users from concerns about flow of control or schedule of program execution events. User's specifications are free from implementation details. Thus they are much shorter (about 3 to 5 times) and easier to comprehend than the corresponding procedural programs. The MODEL translator generates efficient, machine independent object programs. It can be easily adapted for cross generation of code for specialized hardware. It provides global checking and error recovery procedures. Owing to that software development with the MODEL language proved to be much faster than with procedural languages (cf. Szymanski et al. (1984)). In recent years, the research on MODEL focused on efficiency of the generated programs (cf. Szymanski et al. (1986)) and on creating programming tools for parallel/distributed environments (cf. Prywes and Szymanski (1985) and Szymanski et al. (1985)).

The MODEL system consists of three components: (i) the compiler, which translates individual specifications into procedural programs; (ii) the configurator, which creates distributed and/or parallel computation from a set of specifications; and (iii) the recently developed timer, which evaluates processing time (delays) in a specification. It, therefore, provides the basic data for a computational load balance analysis. The advantage of using MODEL for load balancing is that flow of control in the object program is generated by the MODEL compiler and this flow of control can be readily used by the timer for performance evaluation. Another advantage is the ease with which MODEL specifications can be modified. Thanks to the nonprocedural and static view of computations, redesigning a system requires merely moving individual equations from one module to another. As presented in Lee et al. (1986), this leads to a new method of

software development in the presence of time constraints.

Software development with the MODEL system consists of the cooperative use of the three above mentioned automatic components, as illustrated in Figure 1.

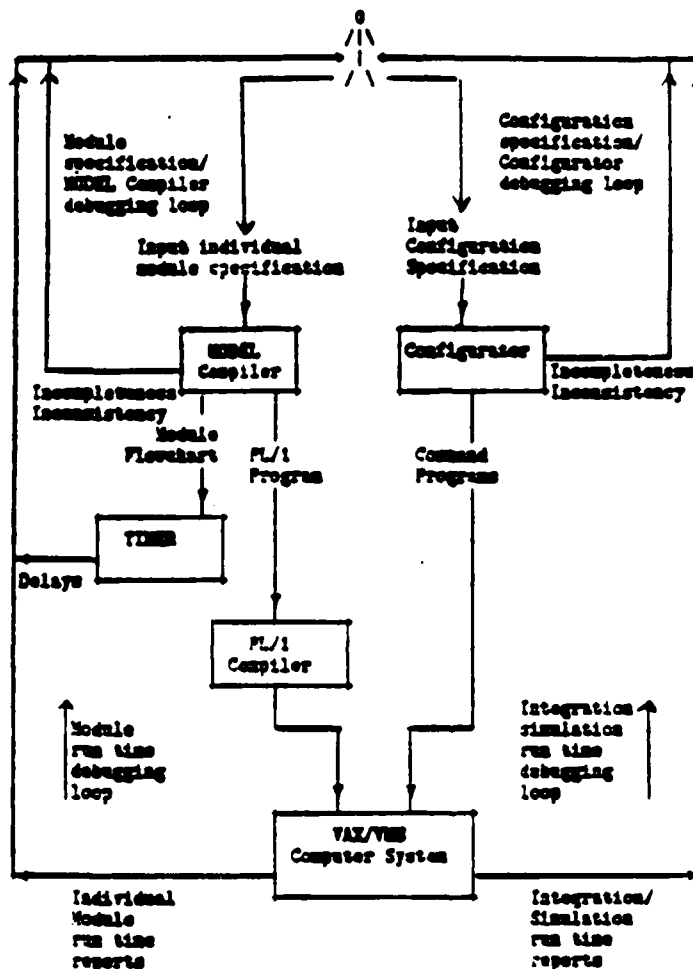


Figure 1. The cooperation of the MODEL system components in developing an application.

Programming in MODEL starts with partitioning an overall computation into *modules* and *files*. Initially, no consideration is given to efficiency. A module's area is delineated, along what appears to the user as the natural functional boundaries. Modules are also candidates for concurrent processing. A *module* consists of a declaration of array variables and a set of equations that relate these arrays to each other. Such module definition in the MODEL language is called a *specification*. Array variables are organized in a tree structure which is denoted as a *file*. A file defined (or *produced*) by one module may be referenced (or *consumed*) by another module. Files may be stored on external memory devices (i.e. disks, tapes) or can be directly communicated between

modules. A computation is viewed on a global level as a graph or a network whose nodes represent modules and files and edges connect modules to files that are consumed or produced by those modules.

The configurator performs the global level design and programming. It accepts a graph of the overall network of modules and files (including assignment of nodes to local or distributed hardware), called a *configuration*, as input. In addition to checking completeness and consistency and producing documentation, it generates command language programs which schedule the execution of modules, maximize parallelism, and set up communication among modules. The compiler performs the detailed level design and programming. It accepts as input a module specification. In addition to checking, it generates a respective optimized program (in object, high-level language). The graph form is convenient for a configuration while the equation and array form is preferable for a module. These two forms can be translated to each other and a configuration is viewed as an extension of module specifications. Finally, the timer evaluates worst case delays in a module between any successive communicating messages. While little, if any, computer skills are needed for using the configurator and compiler, some expertise is necessary to use the timing information to tune a system and, thereby, improve its performance. These reported delays are considered the *service times* of each module.

The MODEL system automatically implements communications between modules by using queues (invisible to the less-expert user). Knowledge of the worst case queue length and service times allows a more-experienced user to evaluate delays and throughput in critical paths of the configuration. Balancing the load on processors may be attained by migration of equations from one module to another or by splitting or fusing modules. Timing information can also be used as a basis for assigning modules to processors.

3. Predicate Analysis in MODEL Specifications

In the MODEL system, a specification is represented by an *array graph*, where a node represents accessing, storing or evaluation of an entire array and the edges represent dependencies among array variables. The underlying graph of elements of the arrays may be derived from the array graph based on the attributes of dimensionality, range, and forms of subscript expressions, which are given for each node and edge in an array graph. A node A corresponding to a m dimensional data or equation array represents the elements from $A(1,1,\dots,1)$ to $A(N_1,N_2,\dots,N_m)$ where N_1,\dots,N_m denote the ranges of dimensions 1 to m , respectively. Similarly, a directed edge represents all the instances of dependencies among the array elements of the nodes at the ends of the edge. The dependencies imply precedence relationships in the execution of the respective implied actions. There are several types of dependencies. For example, a *hierarchical* precedence refers to the need to access a source structure before its components can be accessed, or vice-versa, the need to evaluate the components before a structure is stored away. *Data dependency* precedence refers to the need to evaluate the

independent variables of an equation before the dependent variable can be evaluated. Similarly, *data parameters* of a structure (e.g., defining a range of a dimension in structure) must be evaluated before accessing the respective structure. Therefore, the array graphs represents not only explicit data dependencies resulting from defining a variable in term of others, but also implicit ones implied by the semantics of the language, like hierarchical or data parameter ones. Labeling dependencies by the corresponding subscript expressions (that define mappings between dependent structure elements) enables compact representation of the specification. Array graph compactness and uniformity of representation of different dependencies simplified greatly analysis and optimization procedures of the MODEL system. It is worthwhile to note that flow of control is absent from an array graph, since it is generated later. Therefore, often superficial, dependencies imposed by the way the algorithm is written by the user do not inhabit optimization in the MODEL system.

Data flow analysis usually ignores predicates associated with equations. As the result the following specification poses a problem:

$$\begin{aligned} A &= \text{IF } X > 0 \text{ THEN } 0 \text{ ELSE } B + 1; \\ B &= \text{IF } X > 0 \text{ THEN } A + 1 \text{ ELSE } 0; \end{aligned}$$

Since here A depends on B and B depends on A , the compiler would issue a circular logic message and would create a procedure for solving what appears to it a set of simultaneous equations, defining A and B . A slightly deeper analysis shows that the specification is well formed, yielding directly $A=0$, $B=1$ when $X>0$ and $A=1$, $B=0$ otherwise.

There are other instances where checking of conditions may flag errors that are normally undetected. Those include checking that initial values in recurrent relations and exit conditions in recursive functions are well defined. Similarly, the timer evaluates time delays on every possible path independently of conditions under which any given path can be selected. Therefore, finding, for example, that predicates of two equations are exclusive allows the timer to set $\max(T1, T2)$, not $T1+T2$ as the total time delay, where $T1$ and $T2$ denote time delays of the first and second equation respectively.

This suggests introducing condition information into the array graph. A labeling of edges by the respective predicate, and recognition of equivalence or exclusiveness of predicates that label edges would make the system accept or reject some specifications. This is an open-ended investigation since no algorithmic procedure could solve the problem in its full generality. The main problem is of course to recognize equivalent but not necessarily identical predicates as well as exclusive ones. We believe that partial schemes which we propose below may solve a high proportion of these cases and enhance the system capability.

Procedural verification techniques are usually based on propagation along control flow edges (cf. de Bakker (1980), Jones (1980), Manna (1974), or Reynolds (1981)). Thus, the standard techniques of computing the weakest precondition and strongest post-condition rely on the propagation of logical information backward and forward in

the control flow graph. In an analogous way, it is possible to develop a "proof theory" corresponding to logical information propagation across an array graph. By propagating logical information backward and forward, we are able to check consistency between separately specified constraints on disjoint variables. We are also able to relate different variables used in predicates to source data elements. Therefore, we can rewrite all the predicates of a specification in terms of a uniform set of source variables: i.e. we are able to normalize them. This very important property of normalization results from nonprocedurality of equational specifications. Namely, each variable is uniquely defined and cannot be redefined in a specification, so it is not dependent on the stage of program execution.

Predicates in specifications consist of Boolean terms connected by Boolean operators. Almost exclusively, those Boolean terms are formed from arithmetic inequalities, typically involving subscript expressions. Thus, to assert the relations of different normalized predicates to themselves, we need a theorem proving subsystem in the domain of arithmetic inequalities (cf. Bundy (1983)). We decided to use the Sup-Inf decision procedure for an area of real number arithmetic. It was originally developed for natural numbers by Bledsoe (cf. Bledsoe (1974)), although it does not constitute a decision procedure for the corresponding area of natural number arithmetic. The whole of real number arithmetic is known to be undecidable. We are going to carve out a decidable subpart by allowing only the additive functions, i.e. by excluding trigonometry, exponentiation, etc. Our experience with the MODEL language shows that predicates in equations are almost exclusively limited to additive functions.

The idea of the Sup-Inf method is to show that a negation of a conjecture is unsatisfiable because one of its constants cannot be assigned a type. The negated conjecture is put through a series of normal forms, the last of which assigns a type to each Skolem constant. The Sup-Inf method works with the negation of the conjecture in disjunctive normal form and proves the conjecture if it is unable to assign a type to one Skolem constant in each disjunct. The method can be extended to formulae with free variables by adding a technique for eliminating such variables. The technique eliminates variables from the conjecture using interpolation and reduces the conjecture to a variable free formula, which is then handled by Sup-Inf (cf. Bledsoe (1980)). Extension allowing proper Skolem functions is also possible and leads to a decision procedure for formulae allowing both universal and existential quantifiers at any level. Unfortunately, all known decision procedures for such extension (called Presburger Natural Arithmetic) are very inefficient (cf. Cooper (1972), Shostak (1979)). Therefore we will investigate what types of inequalities are the most commonly used in MODEL conditions and how to modify the Sup-Inf method to include the proper subdomain of arithmetic.

4. Conclusions

Definitional languages offer a new approach to the software development problem. They relieve users from concerns about efficiency and implementation details and provide better checking facilities than procedural languages. They can therefore lead to a

dramatic improvement in software productivity. These promises are however contingent upon developing a language processor capable of generating high quality code from definitional specifications.

In this paper a novel technique of conditional equation analysis is proposed. It is based on application of theorem proving technique for global predicate analysis in equational specifications. It leads to more accurate estimates of time delays, deeper specification verification, and finer identification of simultaneous equations. An initial implementation of predicate analysis in stack structure definitions has resulted in significant optimization of object programs generated by the MODEL translator (cf. Szymanski and Prywes (1986)).

5. References

1. E.A. Ashcroft and W.W. Wadge. "Lucid, A Nonprocedural Language With Iteration." *Comm. ACM*. Vol. 20, No. 7, July 1977, pp. 519-526.
2. J. Baron, B. Szymanski, E. Lock and N. Prywes. "An Argument for Nonprocedural Languages." In "The Role of Language in Problem Solving I." R. Jernigan, B.W. Hamil and D.M. Weintraub (Editors). Elsevier Science Publishers (North-Holland). 1985. pp. 127-145.
3. W.W. Bledsoe. "The Sup-Inf method in Presburger Arithmetic." Memo ATP-18. Math. Dept., U. of Texas. 1974.
4. W.W. Bledsoe and L.M. Hines. "Variable Elimination and Chaining in a Resolution-Based Prover for Inequalities." Memo ATP-56a. Math. Dept., U. of Texas. 1980.
5. A. Bundy. "The Computer Modelling of Mathematical Reasoning." Academic Press. 1983.
6. D.C. Cooper. "Theorem Proving in Arithmetic without Multiplication." *Machine Intelligence*. Vol. 7, pp. 91-99, Elsevier. New York. 1972.
7. J.W. de Bakker. "Mathematical Theory of Program Correctness." Prentice Hall. New York. 1980.
8. C.M. Hoffman and M.J. O'Donnell. "Programming with Equations." *ACM Trans. on Programming Languages and Systems*. Vol. 4, No. 1, January 1982, pp. 83-112.
9. C.B. Jones. "Software Development, A Rigorous Approach." Prentice Hall. 1980.
10. Z. Manna. "Mathematical Theory of Computation." McGraw-Hill. New York. 1974.
11. N. Prywes and B. Szymanski. "Programming Supercomputers in an Equational Language." First International Conference on Supercomputing Systems. IEEE. New York. St. Petersburg, Florida. December. 1985. pp. 37-45.
12. J.C. Reynolds. "The Craft of Programming." Prentice Hall. New York. 1981.
13. J.A. Robinson. "Logic Programming - Past, Present and Future." *New Generation Computing*. Vol.1, No.2, 1984. pp. 107-124.
14. R.E. Shostak. "A Practical Decision Procedure for Arithmetic with Function Symbols." *JACM*. Vol. 26, No. 2, pp. 351-360. 1979.
15. B. Szymanski, N. Prywes, E. Lock, A. Pnueli. "On the Scope of Static Checking in Definitional Languages." *Proc. of the ACM Annual Conference*. San Francisco, Calif. Oct. 8-10, 1984. ACM. New York. 1984. pp. 197-207.
16. B. Szymanski and N. Prywes. "Efficient Handling of Data Structures in Definitional Languages." *Science of Computer Programming*. 1986. to appear.
17. J. Tseng, B. Szymanski, Y. Shi and N. Prywes. "Supersystem Programming with the Model Equational Language." *IEEE Computer*. Vol. 20, No. 2, February. 1986.

AN INVESTIGATION OF PARALLELISM IN RULE BASED SYSTEMS

Steven D. Raney
David A. Marshall
Martin Marietta Aerospace
P.O. Box 179, Denver, CO 80201

ABSTRACT

For many applications rule-based programming languages are too slow. We are studying the use of parallel architectures to increase overall performance of a logic programming interpreter. Using a tightly-coupled multiprocessor, we have found that substantial performance increases can be obtained, and the parallelism inherent in logic programs can be easily exploited.

1.0 INTRODUCTION

Rule based programming languages are popular for AI, cognitive modelling, expert systems, and databases. Production systems, for example, have been used for medical diagnosis [1], computer configuration [2], and spacecraft payload scheduling [3]. Logic programming languages have been used for planning [4], natural language [5] and prototyping database systems [6]. DARPA's strategic computing initiative predicts the existence in the very near future of expert systems containing 30,000 rules requiring the execution of about 12,000 rules per second as opposed to 50 per second for current systems. Performance increases can be obtained through the use of compiler and optimization techniques[7,8,9], faster device technology, special purpose hardware [10,11] and parallelism[12,13,14]. These techniques are by themselves unlikely to produce the necessary performance increases, therefore, parallel execution of expert systems [15,16,17] must be in combined with these other techniques.

Parallelism in rule-based systems is easily identified. These systems consist of a set of facts or assertions, and a set of rules; new information is derived through the repeated application of rules to facts. At any moment, more than one rule may be applicable and these often define alternative solutions. Conventional systems choose one of these rules according to some fixed strategy and apply the rule resulting in new information. The construction of the set of alternatives can be performed by concurrently matching rules with facts; this is called production-level parallelism, or rule-level parallelism. The parallel application of this set of rules is called search level parallelism.

For the remainder of the paper we focus on a specific type of rule-based system, logic programming. Section 2 briefly describes logic programs and the parallelism in them. Section 3 gives a high level description of the architecture we are studying, the BBN Butterfly. Section 4 describes some preliminary studies in rule and search level parallelism, and section 5 conclusions.

2.0 LOGIC PROGRAMS

A logic program consists of a set of assertions or facts and a set of rules or axioms in horn clause form [18,19]. Figure 1 shows a simple program describing the relationships between scholarships and students. Rule 1 may be informally read as "X is entitled to a scholarship if X is a student, X's gpa is Y, Y is greater than 3.4, and X is a senior."

```

give-scholarship(?x) :- student(?x) & gpa(?x,?y) & greater(?y,3.4) & senior(?x)
student(tom)
student(jane)
gpa(tom,3.8)
gpa(jane,3.6)
senior(tom)
junior(jane)

```

```

query: give-scholarship(?x)

```

Figure 1

Logic programs can be executed using a particular form of resolution called SLD-resolution [19], which we illustrate with the following example. A program is driven by a query, or goal, such as: "find an object Z such that Z is entitled to a scholarship." The solution to this query can be viewed as search of a proof tree. Figure 2 shows a proof tree for this program and goal. The goal matches with rule 1 producing the new goal find a student such that the student's gpa is Y, Y is greater than 3.4, and the student is a senior." For the original goal to be satisfied, this conjunction of new goals must be satisfied. The new goal matches with two facts producing two new goals, one of which results in a solution {X = tom}.

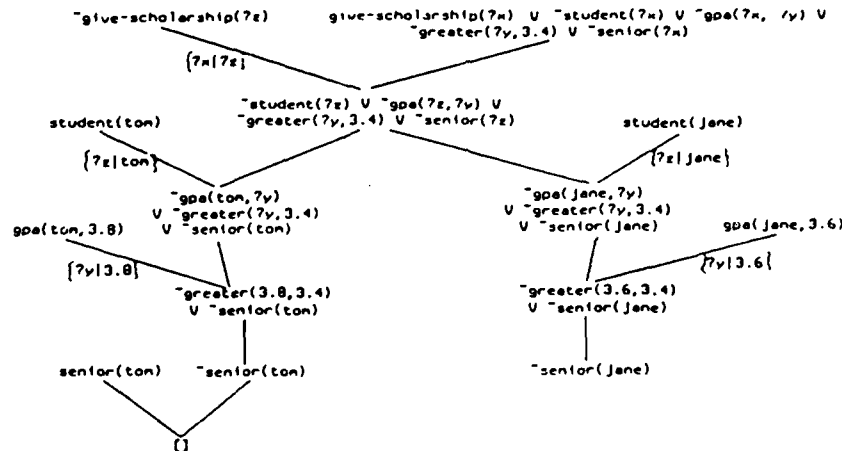


Figure 2

Inspection of the proof tree reveals that parallelism can be exploited in several places, most notably in the match of a goal to rules and facts (rule-level) and the search of resulting new goals (search-level).

2.1 Subrule parallelism

Subrule level parallelism, or intra-rule parallelism [16], refers to parallel unification of a goal and rule. A task is created for each pair of arguments that must be matched. This level is characterized by a high amount of communication and synchronization between tasks. Analysis [10] indicates that, on the average, there are three arguments to be unified, and that two of them share variables making software implementation of unification unprofitable. Further studies [10,11,20] indicate more efficacy from special purpose hardware for unification than implementation on general purpose multiprocessors.

2.1 Rule parallelism

Rule level parallelism, or production level parallelism, refers to parallel match of a goal with multiple facts or rules. If a task is created for each match, this level is

characterized by no communication between tasks, no synchronization, and contention between tasks for goals. This task independence makes rule-level parallelism a good candidate for parallel implementation. The amount of parallelism varies from program to program from little in some such as *append* to very much in database applications where queries are satisfied by many records.

A study of six OPS5 expert systems [15], showed an average of about 35 rules that could be matched in parallel. Other research [10], showed this to be about 10 for several Prolog programs. It is questionable, however, whether these results extrapolate to other expert systems because the amount of parallelism depends on the application. In large databases, for example, particular records can occur in the thousands, and one might observe flurries of massive parallelism for some queries, to very little in others. It may be the case that new semantics for parallel rule-based systems will result in programs with more parallelism than existing programs written for uniprocessors.

An important issue is that of task size. At this level, we define task size as the number of goal-to-rule matches each task performs. The amount of work in a particular match varies as in, for example, $p(X)$ with $p(\text{cons}(1,\text{nil}))$ and $p(X,Y,f(X))$ with $p(a,f(a),f(a))$. Decomposing a task into less than one match results in subrule parallelism. Optimal task size depends on features of the particular architecture such as task creation overhead, frequency and cost of task communication.

2.2 Search Parallelism

Search level parallelism refers to parallel search of the proof tree. Search level parallelism consists of AND and OR-parallelism [14], and many different implementations have been proposed and studied [12,21,22,23].

2.2.1 OR-parallel search

Since alternate branches represent alternate solutions, they can be searched with little or no communication between paths. Variable bindings and modifications to the knowledge base produced by separate search paths must be kept separate. Combinatorial explosion of the proof tree necessitates scheduling heuristics in order to focus on promising paths.

Similar to rule level parallelism the amount of OR-parallelism varies from application to application. If a program contains many rules with the same head, parallelism will be high. Even if only a single solution exists, speedup can be obtained from OR-parallel search since this takes the place of the depth-first search with backtracking used by most systems.

2.2.2 AND-parallel Search

In AND-parallel search conjuncts in a goal (nodes on a path) are solved in parallel. Synchronization and communication is necessary if variables are shared between conjuncts. In figure 2, for example, $\text{GPA}(\text{jane},Y)$ and $Y>3.4$ must agree on a value for Y . Furthermore, $\text{GPA}(\text{jane},Y)$ should be solved first since the set of values it could return for Y is much smaller than the set of values greater than 3.4. Solving $\text{GPA}(\text{jane},Y)$ first would thus constrain the search for $Y>3.4$. $\text{Class}(\text{senjane})$, however, can be solved in parallel with the other two conjuncts without any extra overhead.

AND-parallelism is not, in general, detectable at compile time, because it cannot

occur when two conjuncts contain the same unbound variable or when they contain different variables which have been set to the same unbound variable. These bindings occur at run-time therefore AND-parallelism is only profitable if the amount is large enough to overcome run-time checks [24] A different, but less flexible approach, is to provide a syntax to make AND-parallelism explicit in the program [25].

3.0 BBN Butterfly

The Butterfly is a shared memory tightly coupled multiprocessor consisting of n nodes connected by a butterfly (FFT) switch. Our machine has 16 nodes each consisting of 1 Mbyte of memory as well as a M68000 microprocessor (PE) and a bit-slice co-processor called the process node controller (PNC). The PNC handles all memory references for the node.

Each node's memory is partitioned into a globally addressable and locally addressable memory. When a processor allocates memory, it can allocate to its private memory, global memory, or globally mapped local memory. In the absence of contention, accesses to local memory take 2 usec verses 4 usec for a remote memory access.

4.0 Approach

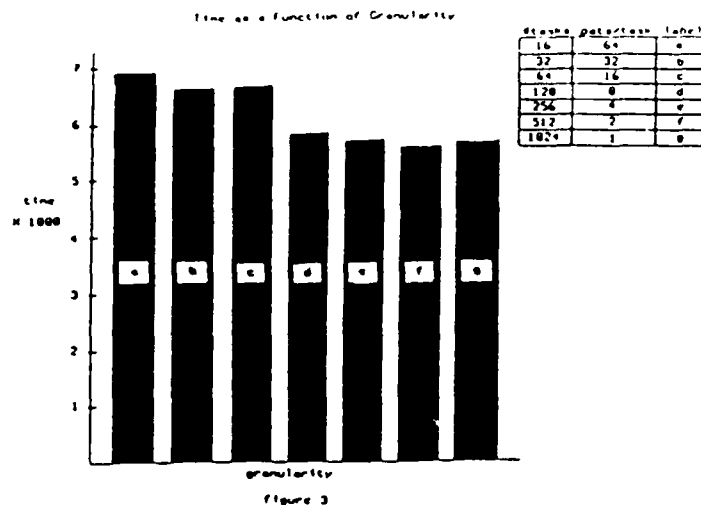
We are interested primarily in rule and search level parallelism because we believe these to be the most promising for performance increases. This is due to the lack of communication and synchronization required between tasks executing at these levels. It is our goal to understand how task granularity and memory allocation affect the performance of an interpreter which exploits these levels.

4.1 Performance of Rule Level Parallelism

Our first approach to parallel unification combines uniform distribution of patterns over shared memory with coarse granularity (few tasks performing many matches). The set of patterns is evenly divided into disjoint subsets, one for each processor. Tasks consist of matching a goal stored in a processor's local memory with each pattern in the processor's assigned subset (stored in shared, possibly remote, memory). The patterns we use are average in size and complexity based on the previously mentioned studies of Prolog programs [10].

In a rule-based system, pattern size varies widely which may cause end-effects for this coarse-grain approach; this arises if a number of tasks match more complex patterns than other tasks. It is therefore important that tasks are created in such a manner that work is evenly balanced across them.

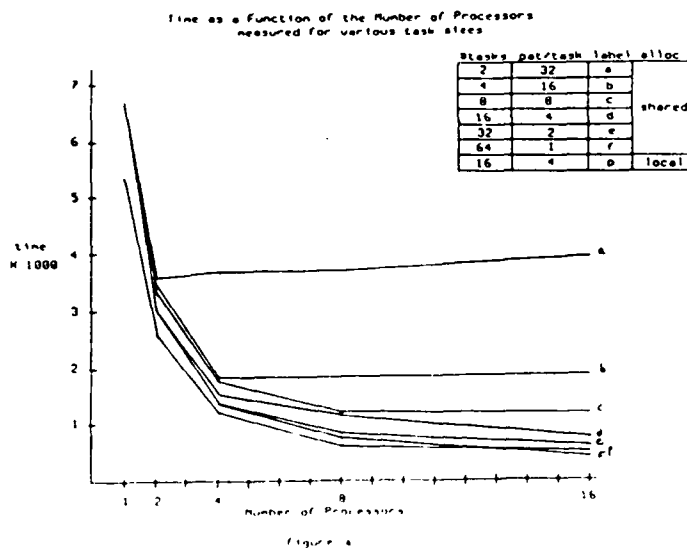
In spite of equal task sizes, the results in figure 3 illustrate that coarse granularity suffers from an end effect due to memory and switch contention.



End-effects can be mitigated by using finer granularity (more tasks performing fewer matches). At the extreme, each task consists of matching the goal to exactly one rule or fact. In a system where patterns vary greatly in size, this fine grained approach allows for load Furthermore, the lack of communication between tasks suggests a very fine grain approach, 1 match per task. Reducing task size further results in subrule-level parallelism.

As expected the very fine grain approach achieves faster runtimes than the coarse grain approach (figure 3). There is a cutoff, however, at which decreasing task size does not appear to improve performance. Decreasing the number of matches per task past 1 introduces too much overhead.

Another way of increasing the throughput of the unification operation is to alter the allocation of rules to processors' local memories. This reduces the time to access memory since no contention can occur, and the operating system does not need to make the reference with exclusive access. The granularity of this approach is dictated by the number of processors. As indicated in the graph in figure 4 the local allocation scheme is no faster than the best shared memory fine-grained approach.



4.2 Performance of Search Level Parallelism

As a preface to addressing search level parallelism we first must fit the results of the

unification study into the context in which it will be used. In the previous experiment the goal was already present in private memory at the time the unification procedure began. For search level parallelism, new goals (paths) are constantly created and must be accessed by all tasks. The goal can be shared or copied into each task's local memory.

We performed a test to assess the affects of copying versus sharing. The results are shown in figure 5. The most noticeable result is that local allocation performed significantly better than global allocation which was not the case in the previous experiments. This is due to the added contention on the goal in addition to contention on rules; local allocation involves only goal contention.

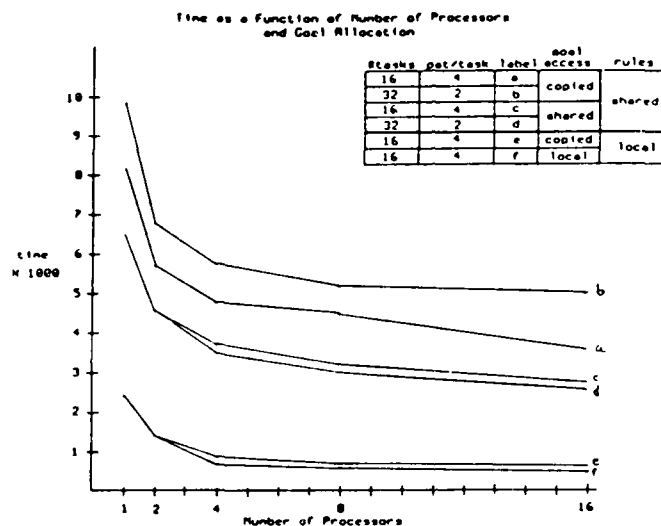


Figure 5

Figure 5 also depicts the fact that regardless of granularity and allocation, sharing resulted in better performance than copying. Due to the dynamic nature of goals, they were represented as tree structures which required traversal for copying. Storing goals in contiguous memory can capitalize on efficient hardware block transfer mechanisms. This, however, requires bounds on the size of these structures.

Our interpreter currently exploits OR-parallelism and a restricted form of AND-parallelism called STREAM-parallelism. STREAM-parallelism refers to pipelining the bindings between conjuncts so that the execution of conjuncts can be overlapped.

A problem occurs when the number of goals (leaves in the tree) grows much larger than the number of PEs. Breadth first search can be performed by creating tasks for every leaf thereby expanding one level at a time. This would be valuable if few solutions were ultimately derivable and the tree quickly thinned out due to failed solution paths. A more complex scheme is to use A* search based on a programmer supplied evaluation function. Another promising strategy is to allow the programmers to weight structures and choose the nodes with the highest values according to these weights. The weights may be assigned to clauses, literals, or symbols, but incorrect weighting can result in infinite failure.

Another problem is the adaption of extra-logical control features such as *assert* and

cut. The assert operator allows the program to modify the database and cut controls backtracking. These extra-logical features often improve the performance of a sequential interpreters, however their non-logical semantics affects parallel interpretation. For example, something asserted by a clause can later be retracted by the clause. In the meantime, it must not affect other search paths. The cut operator is often used to restrict the interpreter to returning a single solution. It prunes the search tree by removing predicates whose heads are similar. This feature is often abused resulting in "if-then" execution of similar rules. These rules then cannot be executed in parallel (unless eager evaluation is used, and the results ordered).

New constructs such as a *one_of* meta-predicate are needed. This predicate would take as an argument another predicate and non-deterministically return a single solution. Furthermore, techniques from concurrent databases such as only updating after an explicit commit will be useful.

5.0 Conclusions

For a 16 node butterfly we were able to show near linear speedup for rule-level parallelism on fine grained global allocation of rules as well as local allocation of rules. The less than linear speedup near the maximum number of PEs was the result of memory and goal contention. This was less of a factor with local allocation resulting in better overall performance with local allocation. This local allocation scheme is similar to the message-passing paradigm.

Using linear speedup as a measure of program performance is not, in general, a good performance measure[26]. This occurs when inefficiencies in the program dwarf the amount of internal communication. If the program is optimized, the internal communications may become a bigger factor and preclude linear speedup. In rule-level parallelism, the program in question is unifying all possible matching rules with a goal. If a more efficient unification procedure were implemented, the result would be a slightly different optimal task size than we observed, but we would still expect the same results due to the independence of tasks.

The less than linear speedup near the maximum number of PEs was the result of memory and goal contention for global allocation and goal contention for local allocation. The upper bound on the number of PEs that can be used (inefficiently much of the time) is essentially dependent upon the number of similarly headed rules and facts which we expect varies widely. Additionally, this involves tradeoffs such as throughput versus efficient use of processors. To achieve maximum benefit from a large number of PEs, however, goal contention must be minimized.

Combinatorial explosion of the search space can be more of a problem for parallel interpreters than their sequential counterparts. This necessitates techniques for allowing programmers to control the search process rather than relinquishing control to an exhaustive search mechanism.

Finally, many of the extra-logical features which are useful for sequential interpreters create problems for parallel interpreters. Control constructs, however, are still desirable and necessary for parallel interpretation. An important area of further work is the development of control constructs which have logical semantics and therefore do not create problems for parallel interpreters.

References

- [1] Shortliffe, E.H., 1976, Computer-Based Medical Consultations: MYCIN, New York, American Elsevier.
- [2] McDermott, J. "RI: The Formative Years." AI Magazine, Summer 1981, pp 21-29.
- [3] Gohring, J., Levy, D., Sauers, R. "EMES: An Expert for Spacecraft Energy Management." 1984 Conference on Intelligent Systems and Machines, Rochester, MI
- [4] Warren, D.H.D. "WARPLAN: A System for Generating Plans." University of Edinburgh Department of Artificial Intelligence Tech Report, 1977.
- [5] Colmerauer, A. "An Interesting Subset of Natural Language." In Clark and Tarnlund (Eds.) Logic Programming, Academic Press, London, 1982.
- [6] Cohen, D. and Hester, T. "A Database Design Analysts Assistant in Prolog." Proc. of the 7th Annual Minnowbrook Conference on Database Machines, 1985.
- [7] Kahn, K.M. & Carlsson, M. "The Compilation of Prolog Programs without the Use of a Compiler." Proceedings of the International Conference on Fifth Generation Computer Systems, 1984, pp 348-355
- [8] Warren, D. H. D. "Implementing Prolog-compiling Predicate Logic Programs", Dept of AI Research Reports No. 39 & 40, University of Edinburgh, May 1977.
- [9] Forgy, C. "On the Efficient Implementation of Production Systems." Ph.D. Thesis Carnegie-Mellon University, 1979.
- [10] Murakami, K. Kakuta, T., and Onai, R. "Architecture and Hardware Systems: Parallel Inference Machine and Knowledge Base Machine." Proc. Int'l Conf. Fifth Generation Computer Systems., Tokyo, 1984, pp 18-36.
- [11] Robinson, P. "The SUM: An AI Coprocessor." Byte, June 1985, pp 169-178.
- [12] Conery, J.S. "The AND/OR Process Model for Parallel Interpretation of Logic Programs." Ph.D. Thesis, University of California at Irvine Technical Report 204, June 1983.
- [13] Stolfo, S. "Five Parallel Algorithms for Production System Execution on the DADO Machine". AAAI August 1984, pp. 300-307.
- [14] Conery, J., Kibler, D. "Parallel Interpretation of Logic Programming." Proc. Conf. Functional Programming Languages and Computer Architecture", ACM, October 1981, pp 163-170.
- [15] Forgy, C.L., Gupta, A., Newell, A., Medig, R., "Initial Assessment of Architectures for Production Systems.", AAAI, Proceedings of the National Conference on Artificial Intelligence, 1984. Austin, TX
- [16] Douglass, R.J. "A Qualitative Assessment of Parallelism in Expert Systems." IEEE Software, May 1985
- [17] Kibler, D., Conery, J. "Parallelism in AI Programs." Proc. Int'l Joint Conference on AI, 1985, pp 53-56.
- [18] Clocksin, W.F., Mellish, C.S. "Programming in PROLOG." Springer-Verlag, Berlin, 1981.
- [19] Lloyd, J. "Foundations of Logic Programming," Springer-Verlag, 1985.
- [20] Tick, E., and Warren, D. "Towards a Pipelined Prolog Processor." Proc. 1984 Int'l Conference on Logic Programming, March 1984, pp 29-41.
- [21] Ciepielewski, A., and Haridi, S. "Control of Activities in the OR-Parallel Token Machine." Proc. 1984 Int'l Conference on Logic Programming, March 1984, pp 49-57.
- [22] Borgwardt, P. "Parallel Prolog Using Stack Segments on Shared-Memory Multiprocessors." Proc. 1984 Int'l Conference on Logic Programming, March 1984, pp 2-10
- [23] Warren, D. H. D. "Executing Distributed Prolog Programs on a Broadcast Network." Proc. 1984 Int'l Conference on Logic Programming, March 1984.
- [24] DeGroot, D. "Restricted AND-parallelism", Proc. Int'l Conf. Fifth Generation Computer Systems., Tokyo, 1984, pp 18-36.
- [25] Shapiro, E. "A Subset of Concurrent Prolog and its Interpreter." ICDT Technical Report, TR-003, 1983
- [26] LeBlanc, T.J. "Shared-Memory versus Message Passing in a Tightly Coupled Multiprocessor: A Case Study." Butterfly Project Report #3, Computer Science Department, University of Rochester, Rochester, N.Y.

Software Support for Heterogeneous Machines

Mario R. Barbacci
Software Engineering Institute
Carnegie Mellon University
2 May 1986

Abstract

We describe a new research effort carried out jointly between the Software Engineering Institute and the Department of Computer Science at Carnegie Mellon University. The objective of the project is to investigate languages, methodologies, and tools for programming computer systems consisting of networks of heterogeneous processors.

Typical users of these notations (and associated support software) will be the developers of real-time, computation-intensive applications such as those contemplated under the Strategic Computing Initiative. In particular, this research is being conducted in the context of the Autonomous Land Vehicle application, running on the Heterogeneous Machine being developed in the CS Department.

This paper provides some background on the nature of the problem posed by these applications, the opportunities presented by the emergence of heterogeneous machines, and the goals of this project.

Introduction

We are all familiar with traditional numerical computation applications, concerned with the accuracy and performance of complex algorithms, operating on simple data structures (e.g. scalars and arrays) implemented in some imperative language (e.g. FORTRAN). The appearance of list-manipulation languages in the late 50's gave rise to symbolic computation applications, concerned with the manipulation of complex data structures (lists and plexes of different kinds) implemented in relatives of Lisp and derivatives of Algol 60. The hardware architectures remained however, relatively constant for a couple of decades and a great deal of progress was achieved in the development of useful programming languages and environments.

We are now beginning to build networks of heterogeneous processors whose users are concerned with allocation of specialized resources to tasks of medium to large size, executing concurrently¹ Heterogeneous machines (e.g. Figure 1) will have general purpose processors, special purpose processors, memory buffers, and switches which can be configured in more or less arbitrary *logical* networks. In addition, these networks are not necessarily static, configured once and for all for a given application; the networks could alter their configuration depending on the needs of the application at any given time.

The Department of Computer Science at Carnegie Mellon University is building a heterogeneous machine as a vehicle for research on high-performance computing. Research is also being conducted in vision processing (e.g. landmark recognition) and machine reasoning (e.g. path planning). These applications depend on the large amounts of computing power that can be delivered by the proposed heterogeneous machine. The research described in this paper is being carried at the Software Engineering Institute and addresses the missing link, namely, the development of languages and methodologies for programming the heterogeneous machine, to exploit the coarse-grain, task-level concurrency available in the applications. Exploring this task-level parallelism is a new direction in parallel processing.

The Nature of the Problem

It is expected that users of a heterogeneous machine will rely on libraries of painstakingly developed procedures to accomplish the common operations in their domain of application. On a high performance engine, with multiple functional units, pipelines, and register sets, these procedures can be very difficult to write but, basically this is within the reach of current compiler technology and programming these engines is not the showstopper.

The major source of complexity in the applications for which the new heterogeneous

¹For our purposes, let's assume that a medium size task granule takes in the order of 100 times a basic synchronization operation. That is, we are not dealing with the minute level of concurrency provided by array processors (e.g. ILLIAC IV) or pipelined functional units (e.g. CRAY), but rather with the scheduling and management of larger chunks of computation, with commensurably larger resource allocation requirements.

machines are being built (e.g. Autonomous Land Vehicle [DARPA83]) does not come from the basic data operations (these are hidden in the node procedures) or the data structures (usually limited to arrays and records). The complexity comes from the communication patterns between the computing elements required to make effective use of the available resources.

The writers of the application programs (e.g. top of Figure 2) must be familiar with the nature of the tasks performed by the processors (nodes in the graph) and the contents of the data queues (links in the graph) in order to program the applications. In general, the tasks will take different amounts of time to complete and the programmer must schedule the arrival of sufficient data at the right time to prevent starvation of nodes, but not so much that queues overflow. Thus, an expert's knowledge of the application is required to select and connect the right resources to achieve some optimal performance. Applications might have additional requirements that might not be directly expressible in terms of nodes, queues, and links. For instance, a requirement might exist that some operation be performed twice as often as some other operation elsewhere in the graph in order to obtain some balanced flow of data. These requirements and constraints are part of the program and must be explicitly indicated.

The efficient utilization of a heterogeneous machine requires therefore support for developing application programs organized as multiple, concurrent, cooperating coarse-grain tasks. The tasks in turn could be more tightly-coupled parallel programs executing on specialized processors such as, for example, systolic arrays. These two programming levels can be separated from each other. The writer of a library procedure that performs some basic computation [e.g. convolution, histograms, etc.] does not necessarily know the context in which the procedure will be used. The procedure executes on a processor that consumes data from input queues and delivers results to output queues. By the same token, the developer of the application does not necessarily know the details of the procedures running on the nodes. The procedures are treated like black boxes or primitive building blocks, with predetermined, perhaps nominal, performance characteristics.

The Nature of the Solution

Suppose that the application programs are represented as graphs, with nodes representing the tasks, and links representing data communication. Programming the task associated with a node involves *intra-node* concurrency, while making the nodes of the graph to work in parallel is *inter-node* concurrency. The intra-node concurrency problem has been thoroughly studied, and there are reasonably mature techniques for writing useful concurrent programs for intra-node computations on special purpose systems. However, the higher level, inter-node concurrency is not as mature or understood; this is the area of interest to us.

As illustrated in Figure 2, a compiler for a task-level programming language will translate the application program into code for a virtual machine. The target "machine language" will consist of commands to be interpreted by a scheduler node. Typical commands include requests for data movements, data transformations, down-loading

code to the computation nodes, invoking task, etc. It is the job of the scheduler to generate the appropriate low level control messages and route them to the processors in the system.

Ideally, neither the language nor the compiler should make assumptions about the structure of the heterogeneous machine, leaving this knowledge to the scheduler. In practice, the programmers may need to know something about the hardware, to perform application dependent optimizations when they choose to do so. We are dealing therefore with (potentially) multiple virtual machine layers, organized in a hierarchy, and implemented by networks of message-passing "smart" resources such as processors, queues, switches, etc. The programmer will develop an application by specifying the operations (i.e., messages) to be carried out by the virtual machine level(s) deemed optimal for the application on hand. The range of abstractions provided by the virtual machines has to be available to the programmer and this has obvious implications in the language design.

In this project we will address the following questions: How much information about the computing engines (nodes) and the tasks running on these engines should be visible? How much information about the machine structure (in contrast to the program structure) should be visible? How much information about the data and control communication infrastructure should be visible?².

Project Goals

The main objective is to develop a specialized programming language for writing distributed programs with coarse-grain concurrency. Suitable constructs will be included in the language for specifying individual tasks, their attributes, relationships between them, and preferred host processors for task execution. In general, language features will be designed in concert with the intended users and the hardware designers. The users will drive the design of the features needed to express task level programs. The designers will provide information about the hardware capabilities. Since the hardware design is taking place concurrently with the design of the language, the former is likely to be affected by the latter (i.e., language features needed to support the applications might require the implementation of appropriate hardware features.)

The task-level, data-flow notation described above appears to be a promising start in this direction, especially for signal processing computations where data continuously flow from input nodes to output ones. There is a reasonably large body of literature on this subject and we are studying a number of existing language models. Given the nature of the problem, dataflow languages such as ID [Arvind78a; Arvind78b] and VAL [Ackerman79; Dennis79] come immediately to mind. However, the applications are

²We distinguish between the *control* communication and the *data* communication networks. The control communication is used by the processors and other resources of the heterogeneous machine to exchange messages of various kinds, as they schedule (or reschedule) the computation tasks. The data communication network, on the other hand, is used to implement the data flow through the machine, and is likely to have higher bandwidth requirements.

likely to require more flexibility than a pure dataflow model: we need to specify computations on streams of data as provided in Lucid [Ashcroft77; Wadge85]. In addition to the data flow operations, the applications require the specification of task synchronization, task control, and graph reconfiguration under a variety of conditions. To satisfy these requirements notations such as path-expressions [Campbell74a; Campbell 74b] might be more appropriate.

Programs in the task-level programming language will be compiled into sequences of task invocation and data communication operations. The first part of the problem to be tackled is the definition of the basic language concepts: the operators and operands used to program the machines at the task level, ignoring the languages and support tools needed to program the basic tasks executing in the computation nodes. The design and implementation of the language and associated tools will be an iterative process, developing virtual machines to execute the task level programs. The first version of the system will provide a simple language, allowing for direct control of the physical resources (the lowest level "virtual machine"). The emphasis will be in obtaining early feedback from the users. Later versions of the system will incorporate additional application requirements (e.g., perhaps better user interfaces.) and multiple virtual machine layers. The abstractions provided by these machines will allow optimizations at the appropriate levels by both the users and the compiler.

By Way of Conclusions

This effort started in January 1986. In the interim, we have been studying existing language models that could be suitable as starting points for the development of a task-level concurrent programming language. At the same time, we have started the design and implementation of a prototype heterogeneous machine (to be operational by the Fall of 1987) and are building a simulator to debug both the language and the hardware design.

Bibliography

[Ackerman79]

W.B. Ackerman, *VAL - A Value-oriented Algorithmic Language Preliminary Reference Manual*, MIT Laboratory for Computer Science, MIT/LCS/TR-218, June 1979.

[Arvind78a] Arvind and K.P. Gostelow, *Dataflow Computer Architecture: Research and Goals*, Department of Computer Science, University of California, Irvine, TR 113, February 1978.

[Arvind78b] Arvind, K.P. Gostelow, and W. Plouffe, *An Asynchronous Programming Language and Computing Machine*, Department of Computer Science, University of California, Irvine, TR 114A, December 1978.

[Ashcroft77] E.A. Ashcroft and W.W. Wadge, *Lucid, a Nonprocedural Language with Iteration*, CACM Vol. 20 No. 7, July 1977, pp 519-526.

[Campbell74a]

R.H. Campbell and A.N. Habermann, *The Specification of Process Synchronization by Path Expressions*, University of Newcastle upon Tyne,

Computing Laboratory, Technical Report 55, January 1974.

[Campbell74b]

R.H. Campbell and P.E. Lauer, *A Spectrum of solutions to the Cigarette Smokers Problem*, University of Newcastle upon Tyne, Computing Laboratory, Technical Report 63, May 1974.

[DARPA83] *Strategic Computing*, Defense Advanced Research Projects Agency, October 1983.

[Dennis79] J.B. Dennis and K.K.S. Weng, *An Abstract Implementation for Concurrent Computation with Streams*, Proceedings of the 1979 International Conference on Parallel Processing, Detroit, Michigan, August 21-24, 1979, pp 35-45.

[Wadge85] W.W. Wadge and E.A. Ashcroft, Lucid, the Dataflow Programming Language, APIC Studies in Data Processing No. 22, Academic Press, 1985.

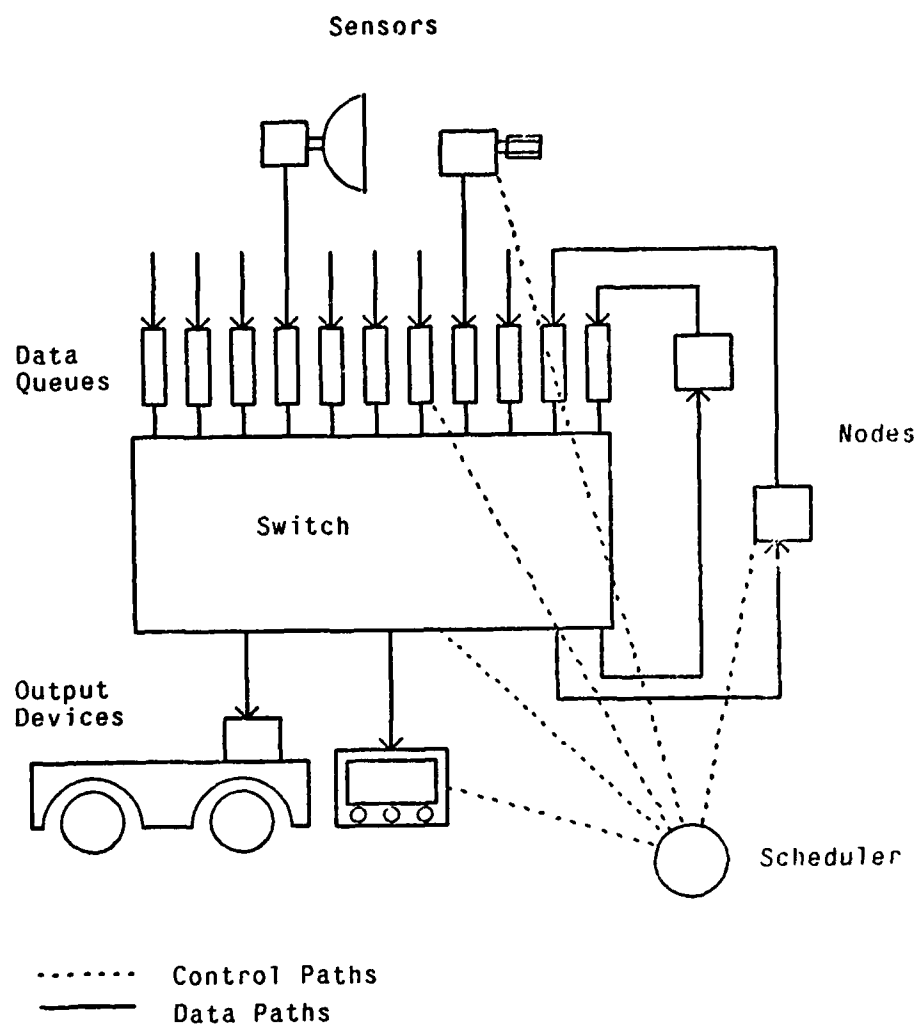


Figure 1 -- A Heterogeneous Machine

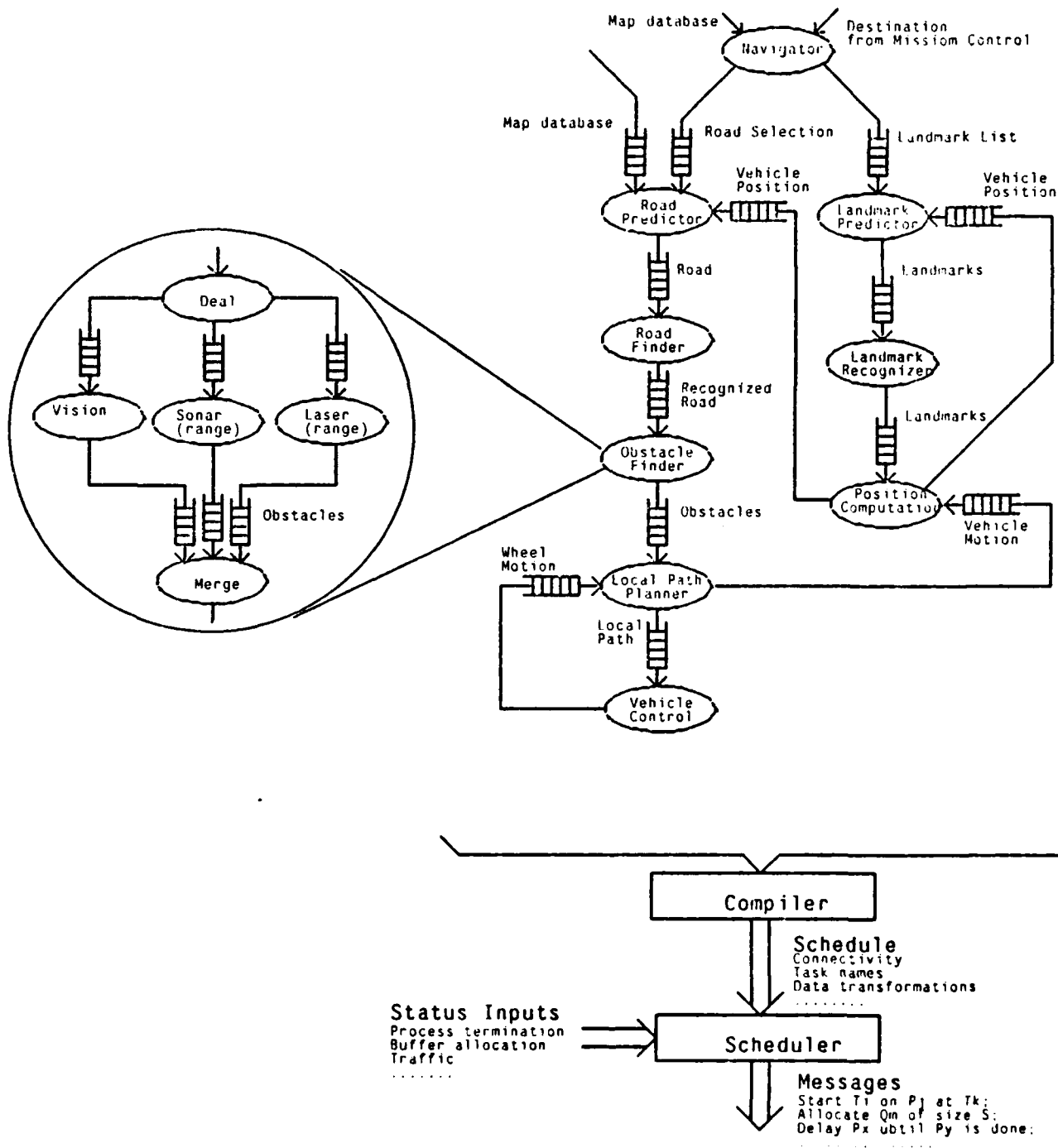


Figure 2 -- Compilation and Execution of a Task-level Concurrent Program

**Session 14A: Interconnection Strategies
& Distributed Computing
Systems**

**Chairperson: L. N. Bhuyan
University of SW Louisiana**

The Case for a Shared Address Space

Edward F. Gehringer

Department of Electrical and Computer Engineering
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-7911

Abstract

Multiprocessor systems that allow processors to access individual words in each others' local memories can efficiently execute a large class of parallel algorithms. Global search, interprocess synchronization, and some forms of transaction processing execute several times more efficiently in a shared address-space environment than in a system that provides only message-passing. Though provided by relatively few multiprocessors, shared address spaces can be implemented quite easily on systems that provide separate communication and application processors at each node.

1. Introduction

Multiprocessors are usually classified as either *tightly coupled*, with main storage shared via a common bus or multiported memory, or *loosely coupled*, communicating via message-passing³. In fact, there exists a middle ground: a shared-address space implemented over the interprocessor communication network. Shared address-space systems have existed at least since the inception of Cm*² in the middle 1970's. The Shared Memory Hypercube⁴ is a recent example. However, relatively few multiprocessors have taken advantage of a shared address space.

2. A Cm* Comparison

The chief advantage of a shared address space lies in its ability to speed up references to scattered words in remote memory. The interconnection structure treats a remote memory reference as, in effect, a special-purpose message to be transmitted to a destination node where a switching processor retrieves or stores a designated word of memory. However, the message is triggered by an ordinary read or write to a mapped region of memory, so explicit invocation of a send operation is not required. On Cm*, remote memory references require from 9 to 35 μ sec., vs. an average instruction time of 8 μ sec. By contrast, a message-passing implementation would require these steps to read a single word:

- The requesting processor prepares a message and invokes a send operation.
- The message is sent over the communication network and stored in a buffer at the receiving end.

- A server process at the receiving end notices the message and responds to it by preparing and sending a return message.
- The return message traverses the interconnection network and is stored in the requesting process's message buffer.
- The requesting processor performs a receive operation and reads the desired word.

There are many sources of inefficiency in this interaction. To begin with, two messages have to be explicitly sent, requiring queue manipulation of the message buffers at both ends. On Cm*, sends and receives cost at least 110 μ sec. each, or approximately 14 times as much as the average instruction. Sending and receiving two messages thus consumes 56 instruction times.¹ In addition, several more instructions must be executed to write the parameters for the messages and interpret their contents. However, this assumes that the server process is running at the destination end when the message arrives. If not, the running process must be pre-empted and the server resumed. Context swaps are notoriously expensive, costing more than 100 instruction times on Cm*'s STAROS operating system. Not only is the requesting process delayed, but the process running at the destination node is suspended for slightly more than two context-swap times.

One can conclude, then, that on Cm* remote-memory references are about two orders of magnitude more efficient than accomplishing the same purpose by exchanging messages. This suggests that global search algorithms that reference a large distributed database would run much more quickly on a shared address-space multiprocessor. But there are other examples of algorithms that benefit from a shared address space.

3. Experimental Evidence

Jones and Ardö^{5,2} compared several different implementations of synchronization primitives for Ada rendezvous. Their benchmark made use of the "server" paradigm, with a single master process assigning "tasks" to a set of identical server processes. Each server process ran on a dedicated processor. Each time a server process finished a task, it was assigned a new task by the master, until there

This research has been supported in part by the OCREAE Program of the National Security Agency under contract number MDA904-86-H-0003

¹This is an oversimplification of the situation on Cm*, because both receives can be begun before the corresponding send completes. However, the conclusion still holds: each message takes a process-to-process interaction time of at least 228 μ sec., or 28 instruction times.

were no more tasks to do. This paradigm is typical of a number of parallel algorithms, including partial differential equations, molecular-motion calculations, and a variety of matrix algorithms.

In the message-passing implementation, the master sends a message to the workers describing each task, and the workers respond by sending a message back each time they finish a task. In the busy-waiting implementation, a worker waits on a shared variable until the master assigns it work, and the master polls the shared variables until it finds a worker that needs more work. A third version, in-line embedded code, has the workers performing procedure calls to indivisibly increment a counter that indexes into a table of task descriptions that are stored in a shared table. No master process is needed in this implementation. The latter two implementations require a shared address space.

When processing time per task was negligible, the busy-waiting approach proved to be about seven times as efficient as the message-passing version. The in-line code implementation has efficiency comparable to the busy-waiting approach for 2 processors, but contention causes it to degrade as the number of processors is increased. Another experiment measured performance when processing time per task varied randomly between 0 and 100 msec. (or 0 to 80 message-interaction times). In this case, in-line embedded code was twice as fast as the other two approaches for small numbers of processors, but its advantage nearly vanished when more than 12 processors were used. One can conclude that a shared address space was beneficial when interprocess interactions were frequent, or when the number of processors was small.

Sindhu⁶ investigated the problem of reliability in multiprocessor operating systems. One requirement was that the actions of operating-system utilities be *restartable* so that an abort in the middle of a utility operation does not damage system integrity. The problem is similar to that of nested atomic transactions in a database, but there are

some differences. An ordinary two-phase commit protocol is too inefficient because several data structures need to be locked in the course of a nested utility call. The locks may not be released one by one as their data structures are successfully updated; instead they must be held until the entire utility call completes successfully. Otherwise an abort could cause partial changes from a utility operation to become visible to other operations. A full description of the solution is beyond the scope of this paper, but it involves making write references to remote memory in the course of a commit. The of sending messages instead would increase the overhead of restartability—already estimated at 30%—to an unacceptable amount.

4. Implementation

The implementation of a shared address space is not particularly difficult, assuming that the distributed switching structure is sufficiently intelligent. For example, Cm^{*} used mapping processors known as Kmaps to map references by one processor to the local memory of another. Cm^{*} had one mapping processor per "cluster" of 10-14 processors. Several newer multiprocessors such as the Ametek System 14 and North Carolina State's B-HIVE system¹ have a separate communication processor (Intel 80186) associated with each main, or "application" processor. The memory-management unit of the application processor can be set to map a certain portion of the virtual address space to the communication-processor "device" (Figure 1). This virtual address consists of three fields, one which specifies the destination processor, another which specifies an object number at the destination, and a third which gives an offset into the object. A directory at the destination node's communication processor supplies the base address of the object in remote memory. The communication processor then references memory at the remote node and passes back the data (in case of a read) or an acknowledgment (in case of a write). The entire sequence is carried out

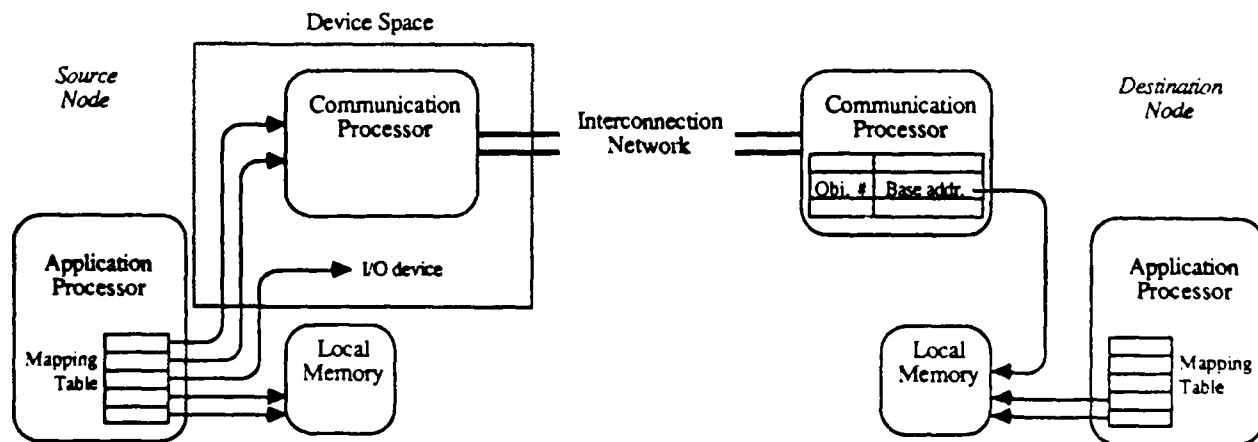


Figure 1: Mapping an Address to Remote Memory

without any explicit system calls (such as *sends*) invoked by either processor.

5. Conclusion

For multiprocessors built from nodes that each have their own private memories, the ability for one node to address another's memory can speed up interprocessor interactions by up to two orders of magnitude. Several algorithms which require fine-grained interprocess communication can profit from this feature. The implementation of a shared address space is easy enough to warrant serious consideration for any multiprocessor with an intelligent communication network.

6. References

1. Dharma P. Agrawal, Winsor E. Alexander, Edward F. Gehringer, Ravi Mehrotra, and Jon Mauney, "B-HIVE project: present and future," *Proc. Austin Conference on Algorithms, Architecture, and Future Scientific Computing*, (Mar. 17-20, 1985).
2. Edward F. Gehringer, Zary Z. Segall, and Daniel P. Siewiorek, *Parallel Processing: the Cm* Experience*, Digital Press, Bedford, MA (1986).
3. Kai Hwang and Fayé A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York (1984).
4. Eugene D. Brooks III, "The shared memory hypercube," UCRL-92479 preprint (1985).
5. Anita K. Jones and Anders Ardö, "Comparative efficiency of different implementations of the Ada Rendezvous," *Proceedings of the AdaTEC Conference on Ada*, pp. 212-223 (October 1982).
6. Pradeep S. Sindhu, "Distribution and reliability in a multiprocessor operating system," Ph.D. thesis, Carnegie-Mellon University, CMU-CS-84-125.

The Next Generation of Hypercube Computers

Trevor Mudge

Dept. Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109

1 Introduction

Massively parallel computers based on hypercube architectures offer an alternative to traditional supercomputers at much less cost. Hypercubes have been discussed in the literature for several decades. As early as the mid-1970's a 256 processor machine was announced by IMS Associates. The processors were Intel 8080's. In 1983 a working hypercube, the 64-node Cosmic Cube, was demonstrated at Caltech [1]. A hypercube of degree n has $N = 2^n$ nodes. The attractiveness of the hypercube over other geometries can be attributed to: 1) the slow (logarithmic) growth in the worst-case internode distance with N ; 2) the slow (also logarithmic) growth in the number of connections to adjacent nodes with N ; 3) the recursive structure of hypercubes—allows multiple users to have disjoint subcubes; 4) the similarity of nodes—there are no special edge nodes as with arrays, for example; and 5) the ease with which trees and meshes of all dimensions can be embedded.

This paper will summarize the developments that one can expect given the expected course of technology in the near future and make the point that software is the major obstacle to the widespread use of hypercube machines.

2 The Present

A number of other machines have been, or are being, developed at Caltech [2,3]. Influenced by this, Intel developed the 128-node iPSC (personal supercomputer) based on 80286/287 nodes. It was the first production machine and was introduced in July 1985. A similar machine, the Ametek System/14 followed.

Currently there are commercial hypercube machines with thousands of processors. The Connection Machine from Thinking Machines has 64K processors arranged in a degree 12 hypercube with each node containing 16 processors [4]. The processors are fairly simple and operate in SIMD mode. The NCUBE/ten has 1024 32 bit processors that operate in MIMD mode. They have a combined potential performance of 500 MFLOPS (million floating point operations per second), although this potential is rarely approached except on special problems. A few thousand processors as complex as state-of-the-art microprocessors is the limit that current technology can support for modest cost air-cooled systems.

The University of Michigan has been a beta site for an NCUBE/ten for the past 6 months and we have had an opportunity to reconstruct the decisions that went into its creation and to perform a preliminary evaluation of the resulting machine [5]. The design process was begun in early '83, and reflects the state-of-the-art at that time. Dominant considerations were packaging constraints, memory integration levels, and custom vlsi integration levels. The result was a node made of only seven components: a custom chip containing the memory interface, internode and I/O channels, and the cpu; and 6 memory chips. The node has a footprint no larger than a playing card. There are 128K bytes of memory per node, and a 10M Hz node can run 1153 Fortran Dhrystones per second (v. 519 for a VAX 11/780) and 445 kilo-Whetstones per second (v. 395 for a VAX 11/780). The chip supports full IEEE 754 floating-point standard arithmetic.

3 The Future

3.1 Hardware

The ability to build reliable inexpensive massively parallel machines has been demonstrated. As a matter of course we can expect faster processors and larger memories. The recent announcement of Intel's iPSC-VX, which includes a high performance vector processing capability at each node, underscores this view. Furthermore, improvements (more speed and higher densities) in designs such as the NCUBE/ten can be expected to keep pace with those in memory chips since each node is dominated by memory.

3.2 Software

Software is a major obstacle. The recent report on the Supercomputer Research Center concludes that the absence of appropriate parallel programming languages and software tools is the single biggest impediment to the successful use of parallel machines [6]. Parallel languages are still in their infancy, and critical development tools such as parallel debuggers are non-existent. This poor support coupled with little understanding of how to express problems as parallel algorithms may limit hypercube machines to special applications. Overcoming this obstacle will be a major challenge.

Acknowledgment This work was supported in part by ARO grant DAAG29-84-K-0070.

4 References

References

- [1] C.L. Seitz, "The cosmic cube," *Comm. ACM*, vol. 28, pp. 22-33, Jan. 1985.

- [2] G. Fox, *The performance of the Caltech hypercube in scientific calculations*, Caltech Report CALT-68-1298, April 1985.
- [3] J.C. Peterson et al., "The Mark III hypercube-ensemble concurrent processor," *Proc. Int'l Conf. on Parallel Processing*, pp. 71-73, Aug. 1985.
- [4] W.D. Hillis, *The Connection Machine*, MIT Press, Cambridge, MA, 1985.
- [5] J.P. Hayes et al., "Architecture of a hypercube supercomputer," *Proc. Int'l Conf. on Parallel Processing*, Aug. 1986.
- [6] Anon, *Report of the Summer Workshop on Parallel Algorithms and Architectures for the Supercomputing Research Center*, Aug. 1985.

A Parallel Computer Based on Cube Connected Cycles

M.J. Chung, E.J. Toy, A.A. Lobo

Department of Computer Science & Center for Integrated Electronics,
Rensselaer Polytechnic Institute, Troy, New York

Keywords: Cube Connected Cycles, Network Emulation, Parallel Computing

1. Introduction

Currently, there are many issues facing designers of parallel computing systems. The development of an architecture independent programming environment and selection of an appropriate implementation vehicle are considered. The main objectives of the programming environment should include maximal exploitation of concurrency, freedom from implementation constraints, transportability/reusability of programs and natural interfacing of the user with the programming language. The concerns of implementing highly parallel systems include the selection of an interconnection network and processing element (PE) to be used in such a network, development of the PE's instruction set and fault tolerance in both the interconnection network and PE. While programs should remain disjoint from the implementation issues, proper selection of implementation details will greatly affect the efficiency of the execution of such programs.

The basis of this work can be found in [1] which outlines a feasible implementation of a general-purpose SIMD parallel computer of 512 PEs, capable of operations on problem sizes of up to 32768 data points. The processors are connected with the Cube Connected Cycles (CCC) network [6]. The salient features of the CCC network are: 1) interprocessor connections are strictly limited to three; 2) highly regular network control; 3) identical PEs; 4) a high degree of pipelining can be achieved; 5) efficient emulation of other networks. The computer is intended for general-purpose, scientific calculations and it is estimated that the computing power falls within the range of 500 to 1000 MIPs. The reader is referred to [6] for the CCC network's details and to [1] for the implementation details.

2. Programming by Logical Networks

We propose a parallel programming environment for SIMD computers based on the specification of an arbitrary, or logical, network of processors. Specification includes a topological description of the network, description of the network's PE, assigning I/O to a set of PEs and defining an algorithm for the PE. A finite set of inter-node and intra-node instructions similar in syntax to Concurrent Pascal defines the algorithm [1]. Translation to an instruction sequence executable on the physical machine is performed automatically by a compiler, which determines the parallelism inherent in the problem. The underlying network should be a general-purpose SIMD computer, capable of emulating any other network with a reasonable time penalty. The Cube Connected Cycles network has been shown to be such a network [3]. Emulation of logical networks whose size differs from the underlying network will also be possible. Programs will be portable between machines since the writing and debugging of such programs is independent of the machine executing the program. Well known structures suitable for a particular computation (e.g., mesh for matrix multiplication) can also be used. By allowing the definition of an arbitrary network, details of the physical network remain hidden, eliminating the need for low-level information. To further aid the user friendliness, a graphical interface will be provided for entering the network's connection pattern and program. Such networks include the tree, mesh, mesh of trees, hypercube and cube connected cycles networks.

3. The Processing Element (PE) for the CCC

A normal data path cycle on a conventional processor consists of the following phases: 1) Operand Fetch; 2) ALU Operation; 3) Result Store. We denote the aggregate of these phases as a major cycle and each phase as a minor cycle. It is our intention to have a processor that is as simple as possible. We claim that providing instructions as major cycles is not consistent with this goal and propose providing instructions as minor cycles. Such instructions will require little decoding or control circuitry while at the same time yielding great flexibility in controlling the network. This approach extends the RISC concept presented in [4]. The proposed instruction set is shown in figure 1 and its associated architecture is shown in figure 2. The PE consists of a register file (RF), an arithmetic logic unit (ALU), an address register, a mask register, instruction queue (IQ), I/O Buffers, a simple instruction decoder and has a data path bandwidth of 16 bits.

In our implementation, 512 PEs will be connected in a CCC network using wafer scale integration (WSI) technology. A floor plan of the PE die is shown in figure 3. Table 1 gives the initial area estimates for each section of the PE's circuits for a scalable CMOS technology with a $\lambda=1.0\mu\text{m}$ (i.e., $2\mu\text{m}$ drawn gate length). The PE die size will be approximately $2.42\text{mm} \times 1.53\text{mm}$. Assuming that a 6 inch diameter (15 cm) wafer is used, there will be only $10\text{cm} \times 10\text{cm}$ of usable area. Given that the inter-PE and global routing consumes 50% of the area, there will be 1346 possible functioning die sites. With a yield of 50%, approximately 670 functioning dice (more than the minimum requirement of 512) will be available after fabrication.

4. Implementation of Highly Parallel Computers

When implementing a large parallel computer, the designer must remain cognizant of how constraints such as physical size (i.e., layout area), PE node degree, instruction broadcasting, interprocessor communications delay and I/O constraints behave as a function of the number of PEs. The CCC network performs well in all of these areas. In [1] it is shown that instruction distribution can be pipelined such that only $O(\log n)$ PEs need to be provided with an instruction at each execution step, while the I/O latency is shown in [6] to also be $O(\log n)$.

The physical size and communication delays can be minimized through the use of Wafer Scale Integration (WSI). Large digital circuits can be fabricated through WSI technology to yield a complete component encompassing an entire silicon wafer. The silicon wafer is packaged intact and can be used as a component in conventional (i.e., board level) design methods. In WSI, the PEs form the die sites on the wafer. The fabrication of these die sites is identical to that of current VLSI fabrication technology. After fabrication, the devices are tested and functioning dice are connected in place. The physical size of the complete system is therefore reduced to the size of the wafer. Communication delays are reduced since the longest path between processors is at most the wafer's diameter. The delays associated with driving signals off-chip as in board level design are reduced to the order of the wire delay within a chip. An effective method must exist to interconnect the functioning devices on the wafer if WSI is to be considered a viable technology [5]. A discretionary wiring technique [2] and a laser link technology [7] are being evaluated for this project.

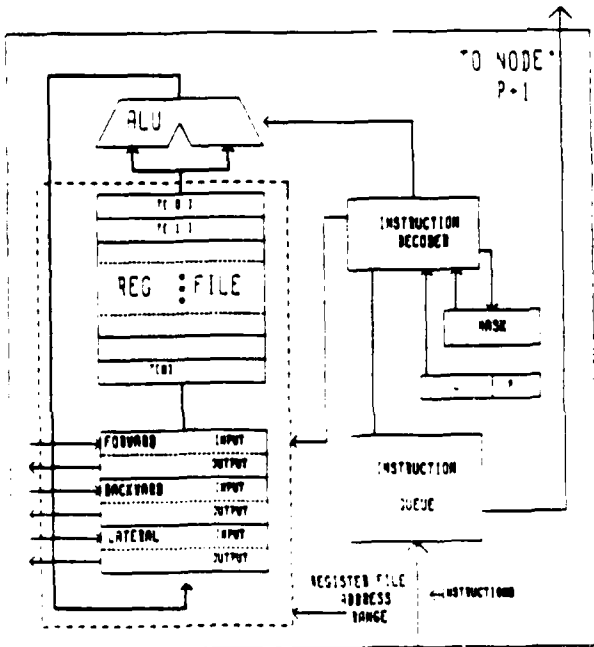
REFERENCES

- [1] Chung, M.J.; Toy, E.J.; Lobo, A.A., "A Parallel Computer Based on Cube Connected Cycles", RPI-CIE Technical Report, March 1986, Rensselaer Polytechnic Institute's Center for Integrated Electronics, Troy, New York.
- [2] Donlan, Lt. BJ et. al., "Computer Aided Design and Fabrication for Wafer Scale Integration", VLSI Design, Vol. VI, No.4, April 1985.
- [3] Gailil, Z.; Paul, W.J., "An Efficient General-Purpose Parallel Computer", JACM, Vol. 30, No. 2, April 1983, pp. 360-387.
- [4] Katevenis, M., "Reduced Instruction Set Computers for VLSI", Doctoral Dissertation, U.C. Berkeley, October 1983.
- [5] Mangir, T.E., "Interconnect Technology Issues for Testing and Reconfiguration of Wafer Scale Integration", Proc. IEEE ICCD, 1984, pp. 127-131.
- [6] Preperata, FP; Vuillemin, J, "The Cube Connected Cycles: A Versatile Network For Parallel Computation", Comm. of the ACM, Vol. 24, No. 5, May 1981.
- [7] Raffel, J.I. et. al., "A Wafer-Scale Digital Integrator", Proceedings of the IEEE ICCD, 1984.

Circuit	Dim. $\lambda \times \lambda$	Area mm^2	% Area	Processing Element Instruction Set			
				Instruction	Operation	Instruction	Operation
Reg File	480x2400	1.183	31.8	LDA Rs	ALU.inA ← Rs	PSA	ALU.out ← ALU.inA
ALU	480x500	0.240	6.46	LDB Rs	ALU.inB ← Rs	PSB	ALU.out ← ALU.inB
I-O reg	480x210	0.101	2.72	STA	ALU.inA ← ALU.out	ADD	ALU.out ← ALU.inA + ALU.inB
Φ.p reg	200x20	0.004	0.11	STB	ALU.inB ← ALU.out	SUB	ALU.out ← ALU.inA - ALU.inB
IQ	480x270	0.130	3.5	STR Rd	RF(Rd) ← ALU.out	AND	ALU.out ← ALU.inA & ALU.inB
I Dec.	200x200	0.04	1.08	NOP	none	SHR	ALU.out ← (ALU.inA)Sh by 1
Mask reg	300x30	0.009	0.24	LDMSK	MASK ← IQ(1)	COMP	ALU.out ← ALU.inA
Routing & I-O pads	-	1.2	32.3	STCCx†	CCx ← 1	FWDTR	BRin* .p; FRout* .(p-1)mod2 ⁿ
				RSCCx†	CCx ← 0	BWDTR	FRin* .p; BRout* .(p-1)mod2 ⁿ
				RSACC†	CC ← 0	LATTR	LRin* .+2 ⁿ .p; LRout* .p
							LRin* .p; LRout* .+2 ⁿ .p
Die size	2420x1540	3.715	100				

Table 1 - Area estimations for PE

† Condition codes are C:carry, V:overflow, Z:zero, S:sign
x can be any of C,V,Z or S



SECRET

Figure 2.

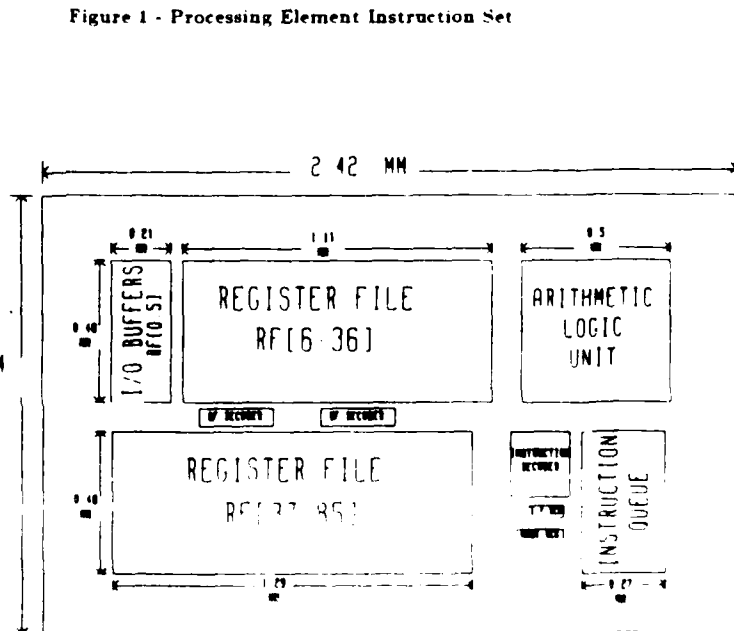


Figure 3. PE Floor Plan

An Analysis of the Reliability of Tree-Structured Interconnection Networks

Ravi Mehrotra and Edward F. Gehringer

Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, NC 27695-7911

Abstract

This paper presents a mathematical model of reliability for tree-structured networks of processors. It assumes that processing elements have an equal probability of failure, and that interconnection links also have a fixed failure probability. From these two probabilities and the number of nodes in the network, a general formula is derived for the average number of nodes remaining connected. It is shown that if an operational tree of a certain size is desired, it is usually more efficacious to improve the reliability of individual processing elements and links than to add more nodes to the network. Indeed, to take best advantage of a large network, the reliability of the individual links must be higher than in smaller networks. The reliability of a randomly structured network is then compared with that of an n -ary tree network. The n -ary tree is shown to be significantly more reliable than the average tree.

1. Introduction

Interest in distributed processing has grown in step with the declining cost of processing elements and advances in networking techniques. These developments have lent added importance to the issue of reliability, because when processing is distributed among many nodes, a failure in any processor or communication link can impede the progress of the entire system. For example, a process at one node may remotely call a procedure at another node to perform some service. The called procedure may, in turn, make use of files or peripherals to which it has exclusive access, or it may call further procedures remotely. Distributed program reliability [1] is a measure of the dependability of such a system. Several other reliability measures have also been developed [1-6].

In this paper we define a very simple measure of reliability for tree-structured networks in terms of the failure probabilities of nodes and links. By studying the entire class of tree structures and deriving statistical averages for their reliability, we gain a quantitative insight into a wide variety of interconnection schemes, and discover how the reliability is affected by changes in network size. Since other networks are simply trees with added communication links, this study also provides a lower bound for the connectedness of general networks.

Consider a distributed system. Its processing elements are prone to failure or malfunctioning. The communication links may also break down, either because of hardware faults or software errors. When processors or links fail, the network may split into more than one subnetwork, with nodes in different subnetworks unable to communicate.

If we select a node in the network, the more nodes that remain connected to it, the more nodes it can communicate with, and, hence, the more reliable is the network topology. Assume that we choose a node randomly from the nodes in a tree-structured network such that each node is equally likely to be selected. Then we can calculate the average number of nodes in the still-connected subtree containing the

chosen node. We will study how this average is affected by the probabilities of node and link failures and the number of nodes in the original tree. We will demonstrate that the size of the connected subtrees is not very sensitive to the probability of failure, as long as the probability of failure is small. We will also show that failure of nodes and links affect a tree more severely as the tree grows larger; for example, for any given probability of failure, a program is more likely to be able to use 20% of the nodes in a small tree than 20% of the nodes in a large tree. Finally, we will show that it is desirable to structure a tree network as a regular n -ary tree, because its reliability is much higher than that of the "average" tree structure.

2. Connectedness after Failures

Assume that the processing elements of a multiprocessor or computer network are represented by the nodes of a graph, and that the edges correspond to interprocessor links. Then the failure of a processor or link is equivalent to the removal of the associated node or edge from the graph.

Neville [7] has shown that labeled trees with n nodes are in one-to-one correspondence with $(n-1)$ -tuples of symbols chosen from the set $\{1, 2, \dots, n\}$. In other words, the structure of any particular labeled tree of n nodes can be encoded by a particular $(n-1)$ -tuple. If a tree T_n is chosen from this set, and some nodes and edges are removed (by failure, for example), the resulting graph is, in general, a forest of disjoint subtrees. Let $\psi_T(S)$ represent the number of nodes in a particular subtree S . If we include the null subtrees consisting of removed nodes, then $\psi_T(S)$ may take on values $0, 1, 2, \dots, n$. As a first step in the analysis, we will find the average number of nodes in such a subtree under the following three assumptions:

- (1) Each tree T_n in the set of labeled trees with n nodes is equally probable. Every tree has, therefore, probability n^{-1} of being chosen.
- (2) A particular node c is chosen from the tree T_n in such a way that any node is equally likely to be chosen. Every node has, therefore, probability n^{-1} of being chosen. After edges or nodes are removed by failure, we let S_c denote the particular subtree (in the resulting forest) that contains node c . If node c itself has been removed, then S_c is defined to be the null tree.
- (3) Nodes and edges are removed from T_n such that any node (or vertex) has probability P_e of being removed and probability $(1-P_e)$ of not being removed. Similarly, any edge has probability P_l of being removed and probability $(1-P_l)$ of not being removed. Removal of a node or edge is independent of the removal of other nodes and edges from T_n , and of the choice of node c . We will use the term "failure" as a synonym for "removal."

We will use these assumptions to derive, first of all, an expression for $E[\psi_T(S_c)]$, the average number of nodes in the subtree S_c .

Define a random variable $\eta_T(S_c)$ equal to the number of failed

edges in the original tree T_n that connected nodes of S_i to nodes not in S_i . Let $n_1^*, n_2^*, \dots, n_{\theta_{T_n}(S_i)}^*$ represent the nodes not in S_i that were adjacent to some node in S_i before the removal of nodes and/or edges from T_n .

Let us form a new tree $T_n'(S_i)$ by taking the original tree T_n and "collapsing" all of the nodes in S_i into a single "giant" node. More formally, $T_n'(S_i)$ consists of the $n - \psi_{T_n}(S_i)$ nodes not in S_i and another node (the giant node) of degree $\theta_{T_n}(S_i)$. $T_n'(S_i)$ is formed by

- removing from T_n the nodes and edges of S_i , together with the $\theta_{T_n}(S_i)$ edges between nodes $n_1^*, n_2^*, \dots, n_{\theta_{T_n}(S_i)}^*$ and some node in S_i , and then
- reinserting these $\theta_{T_n}(S_i)$ edges between nodes $n_1^*, n_2^*, \dots, n_{\theta_{T_n}(S_i)}^*$ and the giant node.

Using the above definitions we will show that

$$E \psi_{T_n}(S_i) = \sum_{l=0}^{n-1} \Pr\{\psi_{T_n}(S_i) = l\} \quad (1)$$

where

$$\Pr\{\psi_{T_n}(S_i) = l\} =$$

$$\begin{cases} P_e & \text{when } l=0 \\ \frac{l!}{n^{n-l-1}} (1-P_e)^l (1-P_e)^{n-l-1} \left[\binom{n}{l} (P_e + P_e - P_e P_e) \right] & \text{when } l=1, 2, \dots, n-1 \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

The total number of trees $T_n'(S_i)$ such that $\psi(S_i) = l$ and $\theta(S_i) = j$ (i.e., whose giant node was formed by collapsing l nodes and has j edges to the rest of the tree) is equal to the total number of trees with $n-l+1$ nodes when the degree of a node is specified to be j . In Neville's codification, the node corresponding to the giant node of $\psi_{T_n}(S_i)$ must occur $j-1$ times in the $n-l+1$ -tuple that represents $T_n'(S_i)$. These $j-1$ entries in the $n-l+1$ -tuple can be chosen in $\binom{n-l-1}{j-1}$ ways. Each of the remaining $n-l-j$ entries in the $n-l+1$ -tuple may be chosen in $n-l$ ways, as there are $n-l$ other nodes. Thus there are $\binom{n-l-1}{j-1} (n-l)^{n-l-j}$ distinct labeled trees $T_n'(S_i)$ such that $\psi(S_i) = l$ and $\theta(S_i) = j$. Since the j edges that join the giant node to nodes $n_1^*, n_2^*, \dots, n_j^*$ may be connected to any of the l nodes in S_i (any of the l nodes that "make up" the giant node), the number of trees T_n which contain a tree $T_n'(S_i)$ such that $\psi(S_i) = l$ and $\theta(S_i) = j$ is equal to

$$\binom{n-l-1}{j-1} (n-l)^{n-l-j} P_e$$

The nodes in S_i can be chosen in $\binom{n-1}{l-1}$ ways, because when one node (node c) is specified to be in S_i , the remaining $l-1$ nodes in S_i may be chosen from the remaining $n-1$ nodes in T_n . Since S_i itself can be formed in l^{l-1} ways, there are

$$\binom{n-1}{l-1} \binom{n-l-1}{j-1} (n-l)^{n-l-j} P_e$$

trees T_n such that $\psi(S_i) = l$ and $\theta(S_i) = j$. This is the number of trees that contain a particular giant node.

We are looking for the probability that the l nodes in S_i remain connected, independent of any failures in the rest of the tree. Since we assume that nodes and edges are removed independently of each other and of other nodes and edges in the tree, the probability that the l nodes and $l-1$ edges did not fail is given by $(1-P_e)^l (1-P_e)^{l-1}$. Now,

$$\Pr\{\text{a particular edge or node or both is removed}\} = P_e + P_e - P_e P_e$$

Therefore, the probability that there is a subtree of size l whose connection to the rest of the tree has been removed by the failure of j edges or attached nodes can be calculated from

- the number of $n-l+1$ -node trees containing a particular giant node,
- divided by the number of n -node trees,
- times the probability that none of the nodes or edges within the giant node have failed, but that all j connections between the giant node and the rest of the tree have been removed due to the failure of nodes or edges.

Thus, for $j=1, 2, \dots, n-l$,

$$\Pr\{\psi_{T_n}(S_i) = l\} =$$

$$\begin{cases} \frac{1}{n^{n-1}} \binom{n-1}{l-1} \binom{n-l-1}{j-1} (n-l)^{n-l-j} P_e \\ (1-P_e)^l (1-P_e)^{l-1} (P_e + P_e - P_e P_e)^j \text{ when } l=1, \dots, n \\ 0 & \text{otherwise} \end{cases}$$

Since the giant node may have been connected to the rest of the tree by up to $n-l$ edges, $\theta_{T_n}(S_i)$ can take values $1, 2, \dots, n-l$. Hence,

$$\Pr\{\psi_{T_n}(S_i) = l\} = \sum_{j=1}^{n-l} \Pr\{\psi_{T_n}(S_i) = l \mid \theta_{T_n}(S_i) = j\}$$

and by substituting for $\Pr\{\psi_{T_n}(S_i) = l \mid \theta_{T_n}(S_i) = j\}$ we obtain

$$\Pr\{\psi_{T_n}(S_i) = l\} =$$

$$\begin{cases} P_e & \text{when } l=0 \\ \frac{(1-P_e)^l (1-P_e)^{l-1} \binom{n-1}{l-1} l^{l-1}}{n^{n-1}} \\ \sum_{j=1}^{n-l} \binom{n-l-1}{j-1} (n-l)^{n-l-j} P_e (P_e + P_e - P_e P_e)^j & \text{when } l=1, \dots, n \end{cases}$$

By substituting $j=j-1$ and using the binomial theorem for the sum on j , we obtain equation (2) after some rearrangement. This completes the derivation of equation (2).

Now consider some special cases that can be derived from equation (2).

- If only processing elements (nodes) fail, not links (edges), then $P_e = 0$ and equation (2) simplifies to

$$\Pr\{\psi_{T_n}(S_i) = l \mid P_e = 0\} =$$

$$\begin{cases} P_e & \text{when } l=0 \\ \frac{l!}{n^{n-l-1}} (1-P_e)^l \left[\binom{n}{l} P_e (n-l+1 P_e)^{n-l} \right] & \text{when } l=1, 2, \dots, n-1 \\ 0 & \text{otherwise} \end{cases}$$

- Similarly, if only links, not processing elements fail, then $P_n = 0$ and equation (2) becomes

$$\Pr\{\psi_{T_n}(S_i) = l \mid P_n = 0\} =$$

$$\begin{cases} 0 & \text{when } l=0 \\ \frac{l!}{n^{n-l-1}} (1-P_e)^l \left[\binom{n}{l} (1-P_e)^{n-l} \right] & \text{when } l=1, 2, \dots, n-1 \\ 0 & \text{otherwise} \end{cases}$$

- If processing elements are not subject to failure, then

$$\Pr\{\psi_{T_n}(S_i) = l \mid P_n = 1\} =$$

AD-A184 949

PROCEEDINGS OF THE WORKSHOP ON FUTURE DIRECTIONS IN
COMPUTER ARCHITECTURE. (U) BATTELLE COLUMBUS LABS

4/5

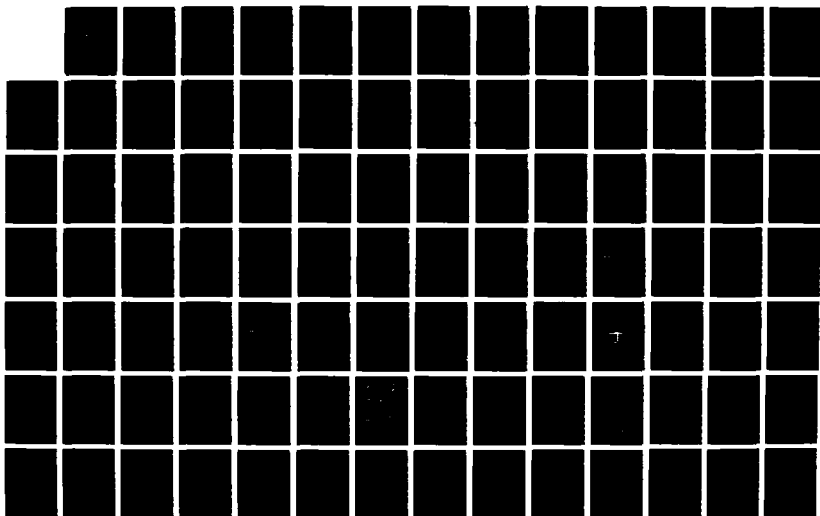
RESEARCH TRIANGLE PARK NC D P AGRAWAL ET AL 30 AUG 86

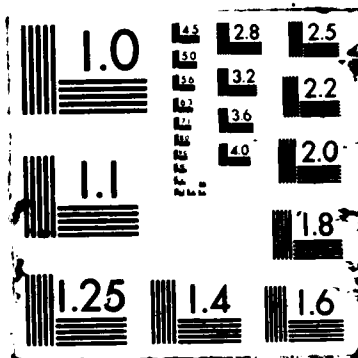
UNCLASSIFIED

AD0-86304-EL DARG29-81-D-0100

F/G 12/5

NL





$$\begin{cases} P & \text{when } l=0 \\ \frac{l!}{n^{n-1}} (1-P)^{2l-1} \binom{n}{l} P(2-P)^{n-l} (1-P)^{2(n-l)-1} & \text{when } l=1, \dots, n \end{cases}$$

These special cases are discussed in greater detail in [8].

3. Connectedness of a Large Network

As the number of processing elements in a network grows large, how is the connectedness affected? Do the still-connected subtrees (giant nodes) grow larger with the size of the network, or is there some threshold beyond which their size levels off? To investigate this question, we will derive a closed-form expression for $\lim_{n \rightarrow \infty} E[\psi_{T_n}(S_c)]$.

We first note that

$$\lim_{n \rightarrow \infty} (n-l-1)^{n-l-1} = n^{n-l-1} e^{-(l+1)} \quad (3)$$

By substituting equation (3) into equation (2), we can verify by a little algebra that

$$\lim_{n \rightarrow \infty} \Pr(\psi_{T_n}(S_c) = l) = \begin{cases} P_c & \text{when } l=0 \\ \frac{l!}{l!} (1-P_c)^l (1-P_c)^{l-1} (P_c + P_c - P_c P_c) e^{-(1-P_c)(1-P_c)} & \text{when } l=1, 2, \dots \end{cases}$$

Since $\sum_{l=0}^{\infty} \Pr(\psi_{T_n}(S_c) = l)$ must equal 1, we obtain

$$\frac{l!}{l!} (1-P_c)^l (1-P_c)^{l-1} e^{-(1-P_c)(1-P_c)} = \frac{1}{(P_c + P_c - P_c P_c)} \quad (4)$$

Note that $P_c + P_c - P_c P_c = 1 - (1-P_c)(1-P_c)$ and then differentiate both sides of (4) with respect to $(1-P_c)(1-P_c)$. The derivative of the left-hand side is

$$\sum_{l=1}^{\infty} \frac{l!}{l!} (l-1) (1-P_c)^{l-1} (1-P_c)^{l-2} e^{-(1-P_c)(1-P_c)} + \sum_{l=1}^{\infty} \frac{l!}{l!} (1-P_c)^l (1-P_c)^{l-1} (-1) e^{-(1-P_c)(1-P_c)}$$

After some algebra, we derive

$$\sum_{l=1}^{\infty} \frac{l!}{l!} (1-P_c)^l (1-P_c)^{l-1} (P_c + P_c - P_c P_c) e^{-(1-P_c)(1-P_c)} = \frac{1-P_c}{(P_c + P_c - P_c P_c)^2}$$

Thus, our final expression is

$$\lim_{n \rightarrow \infty} E[\psi_{T_n}(S_c)] = \frac{1-P_c}{(P_c + P_c - P_c P_c)^2} \quad (5)$$

4. Connectedness of a Specific Regular Tree

A common way of structuring an interconnection network is as a balanced η -ary tree. We will analyze this structure because it is interesting in its own right, and also because it can serve as an example of how specific tree structures can be studied. We will consider an η -ary tree of height one. The results can be generalized to trees of arbitrary height by "collapsing" nodes, as explained in Section 2. An η -ary tree of height one has $n = \eta + 1$ nodes, all but one of which are leaf nodes. We will denote such a tree by T_n^* ; then $\Pr(\psi_{T_n^*}(S_c) = l)$ is the probability that l nodes remain connected in the subtree S_c containing node c after the failure of nodes and edges in T_n^* . As before, we assume that every node has an equal chance of being selected as node c . We need analyze only two cases: where the node c is the root, or where it is a leaf node. These cases are illustrated below.

If c is the root node, then either it fails or it does not fail. It fails with probability P_c ; in this case, l is 0. Otherwise, $l > 0$ and the

$l-1$ remaining live nodes may be selected in $\binom{n-1}{l-1}$ ways. A node remains alive if it neither it nor the link connecting it to the root fails; the probability of this is $\gamma = P_c + P_c - P_c P_c$. If c is the root node, the probability density function is given by

$$\Pr(\psi_{T_n^*}(S_c) = l | \text{degree}(c) = \eta) =$$

$$\begin{cases} P_c & \text{when } l=0 \\ (1-P_c) \binom{n-1}{l-1} (1-\gamma)^{l-1} \gamma^{n-l} & \text{when } l=1, 2, \dots, n \end{cases} \quad (6)$$

If c is a leaf node, $l=1$ if node c does not fail but either the edge connecting it to the rest of the tree or the root node fails. If neither one of these cases occurs, then all of the other $l-2$ live nodes can be reached from c ; these $l-2$ nodes are selected from $n-2$ possibilities. Thus,

$$\Pr(\psi_{T_n^*}(S_c) = l | \text{degree}(c) = \eta) = \quad (7)$$

$$\begin{cases} P_c & \text{when } l=0 \\ (1-P_c)(P_c + P_c) & \text{when } l=1 \\ (1-P_c) \binom{n-2}{l-2} (1-\gamma)^{l-2} \gamma^{n-l} & \text{when } l=2, 3, \dots, n \end{cases}$$

Since

$$\Pr(\text{degree}(c) = n-1) = \frac{1}{n} \quad \text{and} \quad \Pr(\text{degree}(c) = 1) = \frac{n-1}{n}, \quad (8)$$

we can calculate

$$E[\psi_{T_n^*}] = \sum_{l=1}^{\infty} l \Pr(\psi_{T_n^*}(S_c) = l) \quad (9)$$

By substituting equations (6), (7), and (8) in equation (9) and using the binomial theorem we obtain

$$\begin{aligned} E[\psi_{T_n^*}] &= \frac{1-P_c}{n} + (1-P_c) \frac{n-1}{n} (P_c + P_c) \\ &\quad + 3(1-P_c) \frac{n-1}{n} (1-P_c - P_c - P_c P_c) \\ &\quad + (1-P_c) \frac{(n-1)(n-2)}{n} (1-P_c - P_c + P_c P_c)^2 \end{aligned}$$

From this equation we derive

$$\lim_{n \rightarrow \infty} (E[\psi_{T_n^*}] / n) = (1-P_c)(1-P_c - P_c + P_c P_c)^2 \quad (10)$$

Note that the last term of this expression tends to infinity as n grows large. Compare the result for η -ary trees with the connectedness of the average tree derived in Section 2, which tends to a constant for large n . This indicates that bushy η -ary trees provide much better than average reliability.

5. Discussion

Plotting some values of equations (2) and (5) gives an insight into the impact of variations in values for P_c and P_e . Figure 1 plots several curves for the case where only links fail. It shows how the average connected subtree size varies with changing P_c . Three curves are shown, one for 100-node trees, another for 1000-node trees, and the third for infinite-size trees. Figure 1(a) shows that there is a sharp dropoff in average subtree size for $P_c < 0.2$. However, most practical networks have $P_c < 0.1$, and the behavior of the curve in this region is not easy to discern from a linear-scale graph. Figure 1(b), a log-log plot, reveals that most of the decline in average subtree size occurs (for 100- and 1000-node trees) when $P_c > 10$; the trees remain almost wholly connected at link reliabilities higher than this.

It is also clear from Figure 1 that for higher values of n , the average subtree size begins to decline at lower values of P_e (higher reliabilities). Therefore, to take best advantage of large networks, the reliability of the individual links must be higher than in smaller networks.

Figure 2 shows that the phenomenon of node failure has effects similar to those shown in Figure 1(a) for the case of link failure. Figure 3 shows the effect of simultaneous processor and link failures on an infinite-node tree. When both nodes and edges fail in an infinite-size tree, its average subtree size is very nearly equal to the average subtree size of a 100-node tree where only nodes or edges fail. This result can be seen by comparing Figure 3 with Figures 1 and 2.

The analysis also implies that, when a certain average subtree size is needed, it may be more effective to improve processor or link reliability than to increase the size of the network, even by a huge amount. For example, Figure 4 reveals that when P_e is 0.1, an average subtree size of 25 can be obtained from a 100-node tree with P_e of 0.04 but if the probability of edge failure is increased to $P_e = 0.10$, an infinite tree is required.

Section 4 studied η -ary trees and established (equation (10)) that as $\eta \rightarrow \infty$, the average subtree size increases without bound. Figure 5 plots curves for "average" trees superimposed on curves for η -ary trees. For η -ary trees, eleven curves are shown, while for "average" trees, only three curves are shown (because the others would nearly coincide). It can be seen that more nodes remain connected to an arbitrary node in an η -ary tree than in an "average" tree; indeed, for $P_e > 0.1$, an η -ary tree of modest size (150 nodes) has a higher connectedness than even an infinite-size "average" tree.

We have analysed tree-structured networks and demonstrated that the average size of connected subtrees depends more heavily on the failure probabilities of nodes and links than on the size of the network. We have also shown that a specific tree structure, the η -ary tree, has a much better connectedness than an arbitrary tree with the same failure probabilities. Our analysis suggests a method of evaluating the reliability of any specific tree structure. Work is underway to extend it to more general networks, such as X-trees,

hypercubes, and generalized hypercubes (the ALPHA structure).

References

- [1] V. K. Prasanna Kumar, Salim Hariri, and C. S. Raghavendra, "Distributed program reliability analysis," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 1, pp. 42-50, January 1986.
- [2] S. Rai and K. K. Aggarwal, "An efficient method for reliability evaluation of a general network," *IEEE Transactions on Reliability*, vol. R-27, no. 3, pp. 206-211, August 1978.
- [3] K. K. Aggarwal and S. Rai, "Reliability evaluation of computer-communication networks," *IEEE Transactions on Reliability*, vol. R-30, pp. 32-35, April 1981.
- [4] A. Granarov and M. Gerle, "Multiterminal reliability analysis of distributed processing systems," *Proceedings of the 1981 Conference on Parallel Processing*, pp. 79-86, August 1981.
- [5] R. E. Merwin and M. Mirhakak, "Derivation and use of a survivability criterion for DDP systems," *Proceedings of the 1980 National Computer Conference*, pp. 139-146, May 1980.
- [6] A. Satyanarayana and J. N. Hagstrom, "A new algorithm for reliability analysis of multiterminal networks," *IEEE Transactions on Reliability*, vol. R-30, pp. 328-333, October 1981.
- [7] Neville, E. H., "The codifying of tree-structures," *Proceedings of the Cambridge Philosophical Society*, vol. 49, pp. 381-385, 1953.
- [8] Ravi Mehrotra, "Analysis of minimally connected networks for investigating tradeoffs in distributed computers," Master's thesis, University of Hawaii, August 1980.

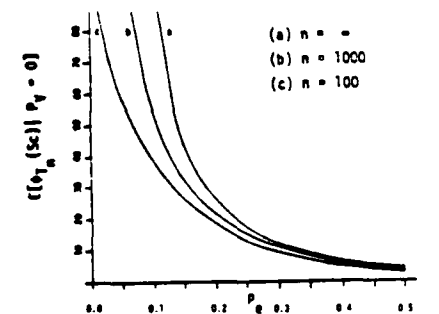


Figure 1(a): Average Number of Nodes Remaining Connected after Failure of Edges

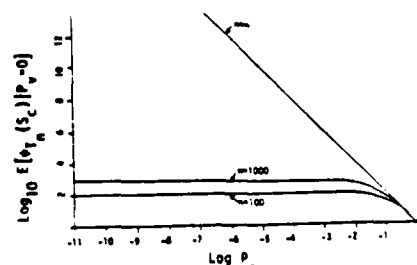


Figure 1(b): \log_{10} of Average Number of Nodes Remaining Connected after Failure of Edges

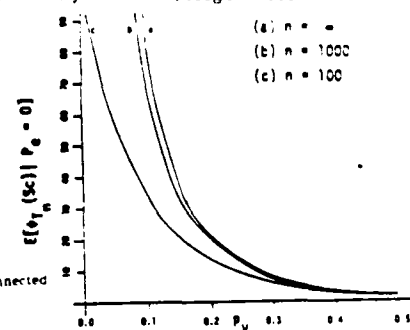


Figure 2: Average Number of Nodes Remaining Connected after Failure of Nodes

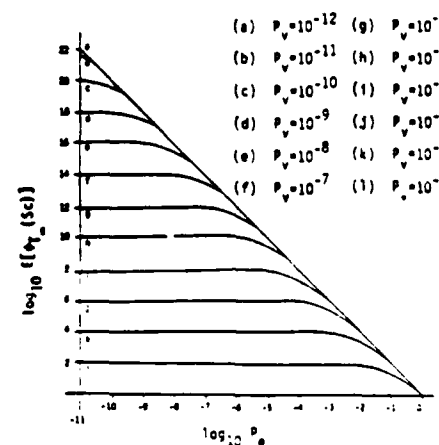


Figure 3: \log_{10} of Average Number of Nodes Remaining Connected after Failure of Nodes and Edges as a Function of P_e for Different Values of P_n

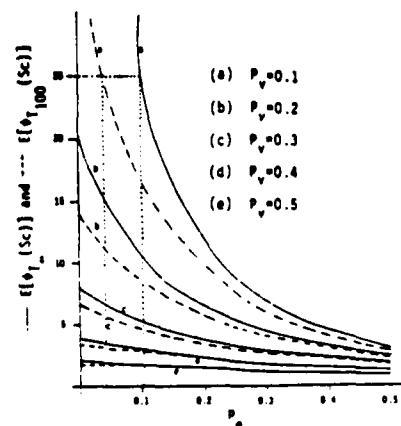


Figure 4: Average Number of Nodes Remaining Connected after Failure of Nodes and Edges as a Function of P_e for Different Values of P_n

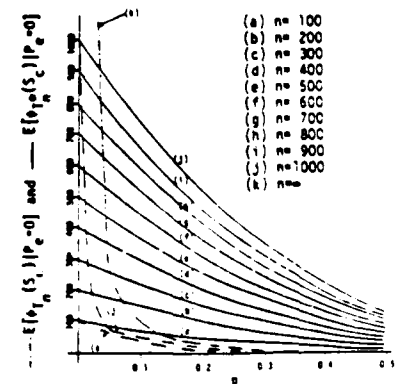


Figure 5: Average Number of Nodes Remaining Connected after Failure of Nodes in n -ary Tree vs. General Tree structure

(This page has been left blank intentionally)

Performance Evaluation of a Loop Interconnection Structure

Edward W. Page
Department of Computer Science
Clemson University
Clemson, SC 29634

Introduction

Highly parallel architectures require efficient, high-bandwidth networks interconnecting a number of processing elements. Loop interconnection structures are of interest because of their advantages of simplicity, modularity, and expandability. As VLSI technology permits the implementation of an ensemble of processors on a single chip, loops become increasingly attractive because they are inherently planar and can be implemented with a high degree of regularity. This paper reports the results of a simulation study of the potential performance of loop-connected multiprocessor systems.

Loop Operation

While a variety of loop configurations have been proposed as interconnection structures [1], this paper focuses upon a packet loop as illustrated in Fig. 1. The packet loop moves parallel word packets from the i th position to the $i+1$ position on each clock cycle. Packets typically consist of a control field, a source field, a destination field and a data field. A processor sends a packet to another processor by writing the packet to an empty slot. Each processor monitors the loop continuously and removes any packet containing its destination address. This type of loop was initially proposed by Pierce [2] and has been incorporated into actual multiprocessor systems [3,4]. Previous studies of loop performance have focused upon the loop mainly in computer network applications [5,6] and not as an interconnection structure to support parallel computations.

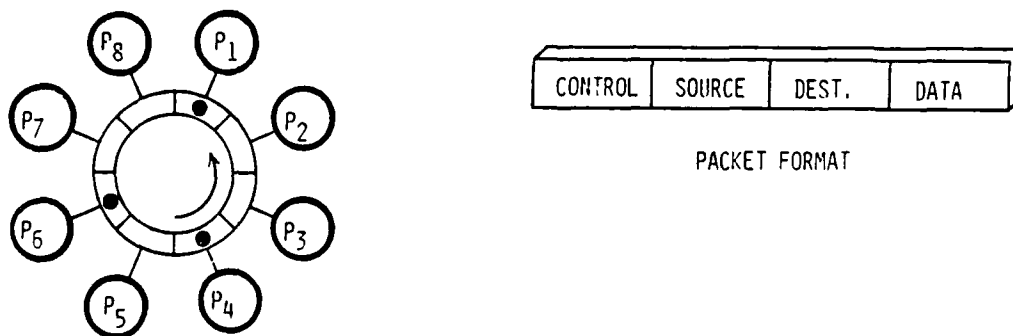
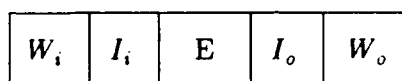


Fig. 1 Packet Loop Concept

Overview of the Simulation

The simulation assesses the potential processing power provided by an ensemble of processors under the communication constraints imposed by a particular interconnection network. Processors are considered to be computers with their own private memories that support multiple processes. Processes communicate with each other through messages that use the interconnection structure. The model assumes that each processor is linked to the interconnection network through an input and an output queue. When data arrives from the network, it remains in the input queue until required by the processor. When a processor produces a data item to send to another processor, it is placed in the output queue until the network can accept it.

A task is modeled as a period of time during which five types of activity might occur as illustrated below:



The symbols are defined as:

- W_i -time spent waiting on arrival of needed input
- I_i -time spent accepting input from the input queue
- E -actual processing time
- I_o -time spent writing results to the output queue
- W_o -time spent waiting for access to the interconnection network

To model interdependent processes, the simulation sends the output of a producer process to the next consumer process that is awaiting data. Should no process need data, a recipient is chosen at random.

Simulation Results

The measure of the efficacy of the interconnection structure produced by the simulation is processing power. Processing power is simply the average number of processors that are in the E state. To establish a baseline for comparative study, exactly the same processing tasks were simulated for both a demand multiplexed bus and a packet loop. The average of the simulation output for several sets of task data is shown in Fig. 2. Here the execution time is uniformly distributed with a mean of 20 bus cycles and the input and output times are 25% of execution time. The loop performance in this case is slightly inferior to the bus until 12 processors are employed and is only marginally better than the bus when 12 or more processors are employed.

On the surface, the results shown in Fig. 2 would seem to indicate that a loop interconnection structure offers little advantage over a conventional demand multiplexed bus. The performance of a multiprocessor ensemble using the bus depends upon the number of processors, the fraction of time the bus is needed and the bus bandwidth. With a loop interconnection structure, however, the physical arrangement of the processors on the loop relative to the communication needs of the individual processes becomes important. A loop connected system will exhibit its best performance when each processor sends its results to its immediate down-stream neighbor. The worst case for the loop results when each processor is required to send its results completely around the loop to its immediate up-stream neighbor. Figure 3 shows the results of the simulation for both the best and worst cases of the loop as compared with the bus.

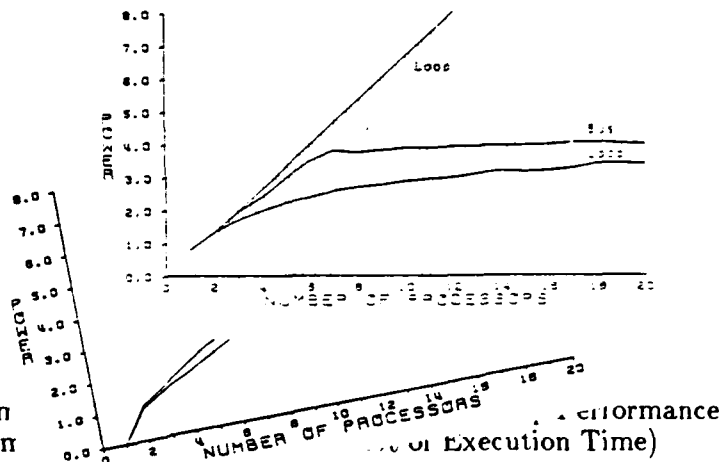


Fig.2 Typical Bus and Loop Perform
(I/O Time=25% of Execution Time)

Conclusion

One cannot state categorically that a loop interconnection structure performs better than a demand multiplexed bus. With a judicious choice in binding of processes to processors, the performance of the loop-connected system can grow almost linearly without bound. However, a poor choice in the assignment of processes to processors in a loop connected system can result in performance that is even worse than the bus in saturation. This study indicates that the loop is most attractive in applications in which traffic patterns are known and can be synchronized.

References

1. Penney, B.K. and A.A. Baghdadi, "Survey of Communication Loop Networks: Parts 1 and 2" Computer Communications vol. 2, pp. 165-180, 224-241, 1979.
2. Pierce, J.R., "Network for Block Switches of Data," Bell Syst. Tech. J., vol 51, no. 6, July/August 1972, pp. 1133-1145.
3. McDonald, W.C. and R.W. Smith, "A Flexible Distributed Testbed for Real-Time Applications," Computer, vol. 15, no. 10, October 1982, pp. 25-39.
4. Kushner, T., A.Y. Wu and A. Rosenfeld, "Image Processing on ZMOB," IEEE Trans. Comput. Vol. C-31, no. 10, October 1982, pp. 943-951.
5. Raghavendra, C.S., M. Gerla and A. Avizienis, "Reliable Loop Topologies for Large Local Computer Networks," IEEE Trans. Comput. Vol C-34, no. 1, Jan 1985, pp. 46-55.
6. Loucks, W.M., V.C. Hamacher, B.R. Preiss and L. Wong, "Short Packet Transfer Performance in Local Area Ring Networks," IEEE Trans. Comput. Vol C 34, no. 11, Nov. 1985, pp. 1006-1014.

ON THE NUMBER OF TASK ASSIGNMENTS IN DISTRIBUTED COMPUTING SYSTEMS

Kang G. Shin

Computer Science and Engineering Division
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109

EXTENDED SUMMARY

In a distributed computing system, a job is usually decomposed into several cooperating tasks and then assigned to processors in order to exploit the inherent parallelism in job execution. The distributed computing system and cooperating tasks can usually be represented by a *processor graph*, $G_P = (V_P, E_P)$, and a *task graph*, $G_T = (V_T, E_T)$, respectively. Here V_T is the set of nodes, each representing a task of the job, and $E_T \subseteq V_T \times V_T$ is the set of edges, each representing intertask communications between the two nodes adjacent to this edge, and V_P is the set of processors in the system and $E_P \subseteq V_P \times V_P$ the set of edges representing communication links between processors. Any two adjacent nodes in the task graph indicate the existence of direct communications between them. (Such tasks are said to be *related* to each other.)

Any two related tasks are required to be assigned to the same processor or two adjacent processors. Henceforth this will be termed the *adjacency requirement*. Using some sort of criterion (e.g., load balancing), the problem of deriving an optimal task assignment is usually formulated as a nonlinear integer programming problem. There are three common approaches to the problem. 0-1 integer programming, use of graph theory, and heuristics [1, 2].

The task assignment problem is known to be NP-complete [3, 4]. In [1], the task assignment problem is formulated as a state-space search problem and then solved by the heuristic algorithm A^* . However, without the knowledge of the size of the state-space, one cannot tell the maximum number of expanded nodes in the A^* algorithm; its complexity is difficult to analyze. Moreover, the knowledge of the number of acceptable assignments can be applied in the state-space search. A search method with this knowledge -- although it may reach a suboptimal goal node instead of the optimal one -- requires much less computation cost in the state-space search and, thus, provides a useful insight into the state-space search problem.

In addition, the knowledge of the number of acceptable assignments and its relation with the processor and task graphs can play an important role in the design of a distributed computing system, when the system's admissibility of incoming tasks is a major consideration. In other words, one needs to determine the system's structure which

¹The work reported in this paper was partially supported by Office of Naval Research, Contract No. N00014-85-K-0122

allows as many acceptable assignments as possible for a given set of cooperating tasks. This will in turn increase the probability that more number of incoming tasks can be assigned to the system while satisfying the adjacency requirement. This knowledge can also be used as a measure of importance of each edge in G_P , i.e., importance of interprocessor communication links.

This work is concerned with the determination of the number of acceptable task assignments, denoted by $N(G_P, G_T)$, rather than task assignments themselves. Specifically, the bounds of $N(G_P, G_T)$ are derived when the task and processor graphs are arbitrary, and then a recursive formula for the $N(G_P, G_T)$ when the task graph is restricted to a tree. If $G_P = R_n$ or $G_P = Q_n$ in addition to the assumption of $G_T =$ a tree, one can derive a non-recursive closed form for $N(G_P, G_T)$. These special cases hold practical importance, since, for example, hypercube multiprocessors are becoming wide-spread in the supercomputing arena.

The knowledge of $N(G_P, G_T)$ is applied to (i) the problem of determining the interprocessor communication link which is of the utmost importance to the system admissibility of incoming tasks, and (ii) the state-space search of the task assignment problem where the search is guided toward an ampler state-space (i.e. with a larger $N(G_P, G_T)$) without computing the associated heuristic function for every possible case. The application approach (ii) is based on the observation that a larger number of assignments implies that unassigned tasks have a better chance to be spread out in the system. When the goal of load balancing is more important than that of reducing interprocessor communications, the likelihood of making a successful guess with the approach (ii) will certainly increase.

REFERENCES

- [1] C. C. Shen and W. H. Tsai, "A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion," *IEEE Trans. on Comput.*, vol. C-34, no. 3, pp. 197-203, Mar. 1985.
- [2] W. W. Chu, L. J. Holloway, M. T. Lan, and K. Efe, "Task Allocation in Distributed Data Processing," *Computer*, vol. 13, pp. 57-69, Nov. 1980.
- [3] R. C. Read and D. G. Corneil, "The Graph Isomorphism Disease," *J. Graph Theory*, pp. 339-363, 1977.
- [4] M. R. Garey and D. S. Johnson, *Computer and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco : Freeman, 1979.

**Session 14B: Architecture and Software
Issues**

**Chairperson: L. Wittie
SUNY at Stony Brook**

REUSE: A RELIABLE UNIFIED SERVICE ENVIRONMENT FOR DISTRIBUTED SYSTEMS

Lionel M. Ni and Thomas B. Gendreau

Department of Computer Science
Michigan State University
East Lansing, MI 48824-1027

1. INTRODUCTION

A distributed system is a combination of a distributed architecture and distributed control algorithms. Distributed architectures cover a wide range of architectures ranging from loosely coupled multiprocessors to long haul networks. In this paper we consider distributed architectures as heterogeneous local area networks (LAN). Such an environment consists of a variety of computing resources, such as diskless workstations, workstations, mainframes, vector processors, and multiprocessors, connected by a high speed (relatively) reliable communication medium. Distributed control algorithms manage the distributed architecture in order to realize the potential advantages of the distributed architectures: availability, reliability, extensibility, and flexibility.

We are currently investigating distributed control algorithms that provide a reliable unified service environment (REUSE). Such an environment presents the distributed system as a unified computing resource which can provide services in a location independent manner. In this paper we discuss some of the concepts that we are using in designing the REUSE system.

2. CLIENT/SERVER GROUP MODEL

The primary form of interaction in the REUSE system is for clients to request that a well-known service be performed. The requested service will be performed by a particular server process. In order to provide reliable service we will encapsulate all server processes that provide a particular service into a *server group* [ChZw85]. Each server group will consist of a set of cooperating servers that manage a set of resources. In general each server in the group will manage a subset of the server group's resources that are similar to the resources managed by the other servers in the group. The redundancy of resources enables the server group to provide more reliable service.

Users will request services through predefined service names. Service names will be mapped into group IDs. Client processes will interact with a server group by communicating with a *local agent* (using the process ID) of the group or by communicating with the group as a whole (using the group ID). Some possible types of server groups include *scheduling agent group*, *file management group*, *mail/communication server group*, and *printer server group*. A scheduling agent group controls the scheduling of processes at different hosts in the system. A file management group will interpret file names and manage files storage devices. A mail/communication server group will do mailing address interpretation, routing, mailbox management, and manage any gateways connected to the system. The printer server group will manage the different printing devices on the system.

As a further illustration of server groups consider the scheduling agent group. Each host in the system will have a local scheduling agent which allocates the processors at that host to the ready processes. The local scheduling agents form the scheduling agent group. These scheduling agents cooperate in order to implement some scheduling policy. Some possible features of the policy could include, load balancing considerations, fairness criteria, and appropriate matching of jobs and host architectures. Scheduling policies may require job migration [NiXG85]. In order to implement some policies, client processes will have to provide the scheduling agent with some characteristics of their jobs. As a simple example consider the case of the server processes of a file management group. One characteristic of these processes is that they should not be migrated because their responsibility is either a local file server agent or is file server managing local devices (or both agent and server) is location dependent. Other characteristics could include expected run time and the type of jobs (vectorized or parallelized).

3. RELIABLE INTERPROCESS COMMUNICATION

The foundation of a distributed programming environment is a set of flexible and reliable interprocess communication (IPC) primitives. The general function of IPC is to create a set of mechanisms that allow location-independent exchange of information between processes in the distributed system. An IPC mechanism consists of various types of *send* and *receive* operations. Various send and receive operations should be provided so that the programmer may exploit potential concurrency in an algorithm and synchronize the communicating processes when it is necessary. The following table lists ten IPC commands that we propose. Detailed definitions and usage of these IPC commands can be found in [NiGe86] and will not be elaborated here.

IPC Primitive Type	IPC Command
<i>asynchronous one-to-one send</i>	Send
<i>synchronous one-to-one send</i>	Send_and_Wait
<i>asynchronous one-to-many send</i>	Multicast
<i>synchronous one-to-many send</i>	Multicast_and_Wait
<i>blocking selective receive</i>	Receive
<i>blocking non-selective receive</i>	Receive_Any
<i>non-blocking selective receive</i>	Cond_Receive
<i>non-blocking non_selective receive</i>	Cond_Receive_Any
<i>non-blocking reply</i>	Reply
<i>blocking reply</i>	Reply_and_Wait

Most of the existing distributed programming languages or known IPC mechanisms support one-to-one interprocess communication. However, in many distributed applications a group of processes are coordinated to solve a single task. These coordinated processes form a *process group* [ChZw85, GeNi85] and may reside in various physical processors. A process in a process group may frequently wish to broadcast a message to those processes in the same group. Thus, a one-to-many send primitive is necessary to allow the message to be sent to a group of processes rather than a single process. The following group primitives are proposed to manipulate groups [NiGe86].

```
Create_Group(group_id, process_id, group_type, group_structure, join_method, status)
Destroy_Group(group_id, process_id, kill_processes, status)
Join(group_id, process_id, timeout, status)
Invite(group_id, host_id, guest_id, timeout, status)
Leave(group_id, process_id, status)
Remove(group_id, process_id, victim_id, status)
```

4. DISTRIBUTED ALGORITHM DEVELOPMENT

A *distributed algorithm* involves a number of processes coordinated to solve a single task. The cooperation of servers in a server group is a special case of distributed algorithms. Distributed algorithms have three important features: *negotiation*, *remote state maintenance*, and *synchronization* [GeNi85]. A negotiation is a set of actions through which two or more processes attempt to reach an agreement. The agreement may be an agreement on a shared value or it may be an agreement to perform some actions. A negotiation can take two general forms: *coordinated* and *peer*. In the case of a coordinated negotiation, one of the negotiators is called a *coordinator* and the other negotiators are called *respondants*. During the negotiation phase, the coordinator provides a central point of control. In a peer negotiation, all negotiators (processes) are equal. There is not a centralized point of control and no negotiator has a complete view of the negotiation. This is a more distributed form of negotiation and a form which is important in distributed algorithms. An example of a peer negotiation is an *election algorithm*.

Another important feature of many distributed algorithms is remote state maintenance. The processes which are cooperating must have some knowledge of each other's state. In order to do this, the processes will periodically report their current state to some of the other processes in the algorithm. It is important to note that, in order for this frequently information is exchanged between the processes, each process will have a slightly different view of the system. The primary problems associated with remote state maintenance are the frequency of remote state reporting and the amount of information that should be reported. A tradeoff exists between the cost of transmitting information and the value of more accurate information. The algorithm designer must identify the essential state of the system and

changes to that information that are significant to the remote processes.

Synchronization issues arise in the areas of initiation of distributed programs, termination of distributed programs, ordering of processes, and controlling access to shared values. In initiating a distributed algorithm, all processes in the algorithm must have identified themselves as ready to begin. Problems arise in deciding how the processes should inform the others that they are ready and how to decide that all processes in the algorithm are ready. In distributed termination, the primary problem is to determine whether all processes have completed a specific phase of the algorithm. Ordering of processes is concerned with identifying that the actions which must be completed before a process can start actually have been completed. Access to shared values must be controlled so that updates to those values are completed in an atomic manner. This problem is further complicated in a distributed system because the shared values may be replicated.

Users may also wish to create distributed algorithms to take advantage of the distributed system. In order to do this the user will create a program which will consist of many processes that will form a *application process group*. The *application process group* has to negotiate with the scheduling agent group in order to get some processor resources. The application process group could specify the type and number of processor resources that it requires (exactly or a range of minimum to maximum requirements). The scheduling agent group would take into account the current load of the system, the number of pending requests, and the priority of the user and respond with the actual resources it is willing to give to the user. The application process group could then configure itself according to the available resources it was granted. A more dynamic system might allow the application process group to negotiate with the scheduling agent group throughout the execution of the application program.

5. RELIABLE SERVER GROUPS

An important consideration in the construction of server groups is the investigation of the tools needed to provide reliable service. By reliable service we mean that the server groups can gracefully handle the loss of resources and servers. This requires coordinated replication of information that is essential to continued (and accurate) performance of service by the server group. The servers in the server group must cooperate in order to identify the loss of servers and to do any reconfiguration required for continued service.

Different types of reliability are required depending on the sophistication of the clients. For example the file servers of a file management group do not expect the scheduling agent group to provide reliable execution of the server processes. It is the responsibility of the file management group to handle the loss of servers due to processor failures. The file management server is considered a sophisticated client whose reliability requirements are beyond the scope of the scheduling agent group. On the other hand the scheduling agent group should take some responsibility for the reliable execution of a user's job. For example if local scheduling agent migrates a job to a remote processor, it should periodically communicate with that processor to make sure that the user process is still being serviced. If the remote processor fails, the local scheduling agent should try to restart the process at another processor. In order to differentiate different classes of clients, the client will have to make their characteristics and requirements known to the scheduling agent group.

REFERENCES

- [ChZw85] Cheriton, D. R. and Zwaenepoel, W., "Distributed Process Groups in the V Kernel," *ACM Trans. on Computer Systems*, pp. 77-107, May 1985.
- [GeNi85] Gendreau, T. B. and Ni, L. M., "A Distributed Game Playing Model for a Distributed Programming Environment," *Tech. Report*, Department of Computer Science, Michigan State University, 1985.
- [NiXG85] Ni, L. M., Xu, C. and Gendreau, T. B., "A Distributed Drafting Algorithm for Load Balancing," *IEEE Trans. on Software Eng.*, pp. 1153-1161, Oct. 1985.
- [NiGe86] Ni, L. M. and Gendreau, T. B., "A Universal Interprocess Communication System for Distributed Computing," *Tech. Report*, Department of Computer Science, Michigan State University, 1986.

R. Pose, M.S. Anderson, C.S. Wallace
Monash University

A Virtual Memory

We are interested in the potential of tightly-coupled multiprocessors to provide general-purpose computing services in an economical and readily expanded fashion. The computer system is viewed as a pool of processors each able to access any of a pool of memory units. The load is seen as a changing population of processes, normally more numerous than the pool of processors (P.U.'s). Some processes may cooperate, sharing subsets of their data, others are completely independent of one another. No explicit commitment of P.U.'s to processes should be visible to users, any more than the commitment of physical memory to data is visible in a virtual memory system.

To allow arbitrary cooperation among processes and indefinite expansion of the system, a uniform, very large virtual memory is defined. To allow secure protection of data and processes from unauthorized interference, access within the virtual memory is controlled by a capability system. The virtual memory contains uniquely-named objects, each being either a data set or a process, and knowledge of a valid capability permits some defined type of access to some (subset of an) object. Because we aim to design a system which can transparently be expanded to encompass networks of similar multiprocessor systems, the naming of objects is universally unique. Each name includes a volume serial number normally identifying a disc volume and an object serial number large enough not to require reuse within the life of the volume. Objects cannot be moved from volume to volume, but volumes may be physically moved from system to system without loss of the identity of their objects.

Basic protection against forgery of capabilities cannot easily be arranged by tagging or segregation of capabilities from other data, since we see it as desirable that data objects containing capabilities be accessible via communication links not necessarily guaranteed to observe the system's protection protocols, and by any agent, human or computer, which can exhibit a valid capability. Capabilities are therefore protected by including in them randomly-chosen 64-bit "passwords". Before any capability is honoured, it is checked against a catalogue of valid capabilities held in or associated with the volume concerned. Since such a check is always required, there seemed little point in attempting to encrypt access rights within the capability itself. Instead, these are held in the catalogue. There may exist many capabilities for an object conferring different combinations of access rights. However, since capabilities are just binary values which may be freely copied and/or transmitted to trusted associates, in practice the number of distinct capabilities simultaneously extant for an object is expected to be small.

It may be thought that the need to consult a catalogue before honouring any capability would impose a heavy overhead. However, in any virtual memory system, system tables must be consulted before honouring any virtual address if only to find the current physical location of the addressed information. As we hold catalogue entries of current interest in a memory cache area associated with page table information, no great additional overhead is imposed.

Window Tables

Clearly, one cannot require rechecking of a capability every time a process wishes to read, write or execute some word of an object. Processes are therefore allowed to validate, and thereafter hold in a "window table", a number of capabilities. With each such window table entry is held location information allowing direct access to the data addressed by the capability. Processor hardware support for

the window table then allows access to the addressed data to be made without further reference to catalogue or physical location tables. Processor instructions typically specify a window table entry number and a byte offset within the (part of the) object defined by the loaded capability. To allow revocation of capabilities, all window table entries are periodically revalidated.

The Intermediate Address Space

Ideally, when a capability is loaded into a window table, the process should be given the information required physically to locate and access any datum within the range of the capability. More realistically, a few page locations might be cached within the processor. While feasible, it would be difficult to maintain the currency of all the page location information held in all the processors, especially information relating to pages in other multiprocessor systems. Instead, every object of current interest in a multiprocessor is mapped into a contiguous block of a single, large intermediate address space, and processors retain in their window tables the intermediate base and limit addresses defined by loaded capabilities. A logical address specified by a window table entry number and offset is translated to an intermediate address which is sent to the memory modules. The memory modules themselves map intermediate addresses into physical page locations.

As the intermediate address space is large (2^{34} bytes in the present prototype) objects can retain their intermediate addresses for long periods, of the order of hours. Physical page locations, which are much more volatile, are held only in the hash-addressed maps within each memory module, and are not duplicated in many caches.

Each multiprocessor has its own intermediate space. References by a process in one system to objects in another are mapped from one intermediate space to the other. To allow physically neighbouring systems to communicate quickly, we have designed an intersystem communication unit which performs this mapping at high speed. It also provides a protection boundary between communicating multiprocessor systems, because one system may establish a mapping of part of its intermediate space into an object in the other system's space only by presenting a valid capability for the object to the other system. For accesses from system A to objects in system B, the unit appears to A as a memory module accepting certain intermediate addresses. Within the unit, these addresses are remapped using a window table containing capabilities for objects in B. Thus the unit appears to B as a processor. Physically, intersystem units are constructed as modules, one of which is connected to each multiprocessor system. Each module can appear as a memory, to accept outgoing access requests, and as a P.U., to effect accesses on behalf of other systems.

Communication among physically distant systems would require use of a conventional network. As such communications are necessarily slow relative to memory speeds, we plan to specify the required accesses by explicit capability/offset pairs rather than try to maintain any intermediate space-to-intermediate space mapping.

The existence, size and values of intermediate addresses are invisible to processes.

Interprocess Communication

The capability-addressed vertical memory allows sharing of data (and of course code) objects among many processes. In addition, a simple message scheme is provided. Every process contains a mailbox into which other processes may place messages, provided they have an appropriate capability. Message sending is defined as a primitive operation on the virtual memory, and direct modification or inspection of mailboxes is not allowed. Each mailbox is of finite size, so processes are expected to inspect and remove incoming messages periodically.

For synchronization purposes, processes may voluntarily suspend until a message is received or a timeout expires. Arrival of a message normally does not interrupt a process. Indeed, we provide no equivalent of conventional interrupts. If a process anticipates the need to take prompt unscheduled action on receipt of a message, it may create another process to wait for such a message. However, a process with appropriate capability can forcibly suspend another process, at least at the expiry of the latter's current time slice.

Since interprocess communication is oriented to processes rather than P.U.'s, and is not defined to cause interruption, no physical communication path among P.U.'s is required for this purpose.

Processor/Memory Communication

Every P.U. must be able to access every memory module. A time-shared bus is a convenient, if limited-bandwidth, means. To minimise traffic on the bus, this design like most similar multiprocessors relies on caches for instructions and data within each P.U. If shared, mutable data is held in these caches, rather elaborate schemes must be used to maintain consistency. Physically, such schemes require either inter-PU communication paths or broadcast signal paths to all P.U.'s or both, and may require every PU to monitor the memory traffic from other P.U.'s. While all these requirements can be met in a bus-coupled system, they result in a rather complex bus protocol, and may limit the bus cycle time to the cycle time of the P.U. cache memory.

We are skeptical of the need to support a large PU access rate to shared mutable data, and do not require inter-PU communication for interrupt purposes. We have therefore chosen to allow shared mutable data to be excluded from PU caches, and by so simplifying the bus protocol, allow a higher bus bandwidth. The present prototype uses a 34-port 32-bit 25 ns synchronous bus, but larger systems of this architecture could use dual unidirectional buses, since traffic is always PU-to-memory or memory-to-PU. The bus technique used has been shown capable of unidirectional operation at 12 ns cycle speed. Further, since no PU or memory requires access to traffic of other P.U.'s or memories, multiple disjoint buses or Omega networks could be used.

I/O Processes

Disc storage for objects is provided via disc controllers which access memory modules in the same way as P.U.'s. Since the demand paging scheme covers all objects, including the analogues of conventional files, no explicit disc access action is defined at the kernel level. Other peripherals are controlled by dedicated processes running permanently on dedicated P.U.'s, and user-level processes effect I/O by communication with these I/O processes. The dedicated P.U.'s are physically attached to a lower-speed bus connected to the main bus by an adaptor, but share the same virtual and intermediate spaces seen by other P.U.'s. They also have private local memory, and can offer services such as editing and command line interpretation.

Status

A prototype using NS32032-based P.U.'s has been operating since January 1986. Most features of the virtual memory are implemented. As the P.U.'s are too slow to use effectively the available bus and memory bandwidth, bit-slice RISC P.U.'s are being designed. A Vax 11/750 is at present used as an I/O processor. Fuller details of the virtual memory interface have appeared in Anderson, Pose and Wallace, 'A Password Capability System', Computer Journal, Vol. 29, No. 1, pp 1-8, 1986. The bus design is described in Wallace and Koch, 'ATTL Compatible Multiport Bus', Computer System Science & Engineering, 1, 1, Oct. 1985.

DIRECTIONS IN COMPUTER GRAPHICS ARCHITECTURE

John Staudhammer
College of Engineering
University of Florida
Gainesville, FL 32611

In the last ten years Computer Graphics has become a large part of every major computer application package. Today's graphics are predominantly high resolution raster display oriented [1.] The relatively common high resolution 1k x 1k 60 Hz refresh display needs a pixel access time of about 15 ns. The High Definition TV systems now being tested in Japan have a 1500 line resolution; thus we may expect that 2k color raster displays will become commonplace in the near future. The size of the displays has also increased: currently 35 inch tubes are available. Much progress has been made in understanding display algorithms and the generation of finely detailed 'realistic' images. One can expect further development in these areas. Computer graphics is used for visualization of designs, of spatial relationships, of feature dependencies, of understanding multi-valued abstract interactions. However, most of the work on image rendering has been directed towards the making of visually pleasing, publishable static images. These are usually made up in large image buffer memories. Two problems are associated with this scheme: the image display rate and the image update rate.

The image display rate is largely fixed for directly viewed pictures. Depending on the ambient light and image brightness, the image needs to be repeated once every 30 to 15 milliseconds [2]; the 60/second update rate now used is likely to remain an acceptable 'standard'. The image update rate will be a compromise between affordable computer costs and needs for visualization. The beauty and clarity of a display image is not the final arbiter for the usefulness of that image (although so judged now from published still images.) When a person needs to know the nature of an object, a great many views are required to form an understanding. These views must be done in a short time; a task not possible with static image generators. Typically the image is computed using a memory in a general-purpose machine and is then transferred to the display generator. In addition to the computational requirements, a data transfer problem must be faced.

In visualizing three-dimensional designs, such as mechanical assemblies, architectural entities, naval and aerospace vehicles, a simple, uniform colored background, a simple light reflection algorithm and relatively crude surface approximations are usually adequate. Studies of the image data structure of a raster scan representations of such scenes show three main features [6]:

1. Long strings of uniform colors, mainly the background;
2. Shorter strings of colors which represent the main features of the design object;
3. Many very short strings, most only a single dot, which give the details of the surface, the minor features.

Taking advantage of these image characteristics one can build relatively inexpensive encoding and decoding machines which will reduce the amount of data required for the description of acceptable images by large factors; a data-compression of 10 to 15 is achievable. Not only is the image buffer reduction significant, but more importantly it is possible to maintain an image flicker-free without recourse to prodigious computer I/O rates. Thus robot manipulator actions can be viewed using only normal input/output channel speeds on machines such as VAX computers. Further reductions in image storage volumes and consequent amelioration in data transfer rates can be achieved by considering the scan-line to scan-line coherence, but more processing will be involved.

Various simple data compaction schemes exist which are particularly useful for managing images arising in mechanical design. Some of the image generation problems are now being addressed by special display management chips [3]. When the image is moving, as in a movie sequence, there is 'motion blur', which may drastically decrease aliasing problems in single images. Special display pipelines will handle specific types of display tasks. Examples are clippers (for object collision detection), element generators (for molecular models) and texture generators (for terrain models and architectural tasks). All of these are 'naturals' for VLSI implementations. However, before any of these can be constructed, the algorithms must be fully understood and thoroughly tested. The optimization for 'acceptable' image quality can be made; this step usually involves an assessment on how much one can 'get away with' in trading motion for clarity of static images.

Finally the computations required for detailed image presentation seem to mandate supercomputer usage [4]. Currently-used algorithms are poorly suited to supercomputers' vectorization requirements. A critical view needs to be taken at re-casting the display calculations into forms more adaptable to vector machines. One of the most compute-intensive tasks in computer-graphics today is the rendering of good-looking curved surfaces by means of detailed calculations of the color-shade of the surface [8], most often by the technique of ray-tracing. A brute-force application of the commonly-used techniques of ray-tracing shows a very high degree of non-uniform calculation steps. Basically each ray will interact with the scene differently: after all, this is the essence of rendering the image in interesting detail. However, a re-thinking of the procedure in terms of a particular supercomputer structure has yielded great savings in computational times and cost, albeit at an expense of significant increases in the number of calculations required and the amount of storage employed [5].

The need to generate good graphic outputs from application programs has spurred the development of specialty display processors and display pipelines. Virtually every major IC vendor has made significant efforts to provide devices with much increased performance. Noteworthy among these is AMD, National Semiconductor, Weitek and NEC. Specialty display systems based on novel architectures are Silicon Graphics's IRIS and tiling engines now becoming available. Gouraud shading can be included in tiling devices which can keep up with real-time 60 Hz displays, usually updated through a double buffering scheme at 30 Hz or

less. Another significant development is the emergence of graphics controllers directed towards video RAMs. Here the familiar BitBlt pixel operations are extended to blocks of pixels, usually referred to as PixBlt operations. As yet, almost all these engines operate on 2-D data, reflecting, perhaps the current graphics standards thinking in CORE and Graphics Kernel Systems (GKS). With the current efforts in PHIGS, much more three-dimensional graphics will become commonplace. Modeling and rendering curved surfaces with dedicated hardware often run into numeric round-off problems and require inordinate numeric accuracy. For a worst-case study of 8k by 8k images of quadratic surfaces the control of accuracy can lead to 8 more than 80 bits in the internal registers [7.] Of course further work will lead to a re-thinking of this process also and to simpler hardware. Even with the worst-case it is possible today to put a few of the quadratic-rendering engines on a single VLSI chip.

With the general trend of decreasing hardware costs the currently-touted devices will spawn standardized architectures for the display task. In turn the display process will have to be re-visited to ascertain how it can be best mapped to high-performance cheap hardware. Moving image sequences can demonstrate that the most beautiful images are not necessarily the most useful ones. We can expect progress in devices just for such images.

REFERENCES:

1. J. Staudhammer, Ed., Special Issue on Computer Graphics Hardware, IEEE COMPUTER GRAPHICS AND APPLICATIONS, January 1986
2. H.R. Luxenberg and Kuehn, Display Systems Engineering, McGraw-Hill, 1972; also J. Staudhammer, Computer Graphics Hardware, Short Course Notes, IEEE Tutorial Week - East, 1985
3. Special Report on Application Specific ICs, in COMPUTER DESIGN, February 1, 1986 issue
4. D.P. Greenberg and J. Staudhammer, Eds., Special Issue on Super-computer Graphics, IEEE COMPUTER GRAPHICS AND APPLICATIONS, July 1987 (scheduled)
5. D.J. Plunkett and M.J. Bailey, The Vectorization of a Ray-Tracing Algorithm for Improved Execution Speed, IEEE COMPUTER GRAPHICS AND APPLICATIONS, August 1985
6. L.R. Schneider, Characteristics of Computer-Generated Raster Images, MS Thesis, Department of Electrical Engineering, University of Florida, December 1985
7. S.T. Kaufmann, Real - Time Interactive Visibility Computation for Quadratic Surfaces, MS Thesis, Department of Electrical Engineering, University of Florida, August 1986
8. T. Whitted, A Processor for Display of Computer Generated Images, PhD Dissertation, North Carolina State University, June 1978

DEVELOPING A STANDARD TAXONOMY OF
SOFTWARE ENGINEERING STANDARDS

JOHN FENDRICH
COMPUTER SCIENCE DEPARTMENT
BRADLEY UNIVERSITY
PEORIA, IL 61625

This paper presents a view of a possible future direction and environment for the development and use of computer software. The paper presents a view of this software engineering discipline and activity within a future environment which is guided and influenced by consensus standards. In particular, a view of a draft standard taxonomy of software engineering standards is presented together with a view of the utility of that taxonomy in classifying standardization efforts in software engineering environments.

A taxonomy can be described as a scheme which divides a body of knowledge and explains the relationships among the pieces and is used for classifying and understanding the body of knowledge. In this case an IEEE Computer Society (IEEE-CS) project, P1002, was authorized by the Subcommittee on Software Engineering Standards (SESS) of the Technical Committee on Software Engineering (TCSE). A key goal among the goals of this project, was to apply this taxonomy concept in order to develop a standard taxonomy of software engineering standards. More specifically the goals of the P1002 project included:

To provide a comprehensive scheme for classifying software engineering standards, recommended practices, and guides.

To provide a framework for identifying the need for new software engineering standards, recommended practices, and guides.

To present a comprehensive scheme for analyzing a set of software engineering standards, recommended practices, and guides appropriate for a given industry, company, project, or particular work assignment.

To present a framework for comparing sets of software engineering standards, recommended practices and guides to support the selection of the most useful set for a particular software product.

Particularly the goals were relative to SESS standardization projects within IEEE-CS, but the goals were seen to have potential application more generally.

The P1002 project produced a Draft Standard Taxonomy for Software Engineering Standards which is currently being ballotted within IEEE. This software engineering standards taxonomy is organized into three parts (1) a partition of standards, (2) a partition of software engineering, and (3) a framework which relates to the two partitions to achieve a partition of software engineering standards. The partition of standards is organized by type of standards, namely profession, product, and process standards. The partition of software engineering is organized by function and lifecycle. The taxonomy framework, as it is applied to achieve the goals of the project, is simply a matrix view in which the two partitions are brought face-to-face.

The thesis of this paper is that we can have an optimistic view of the future of computer software. This optimistic view is based on ideas that,

1. Standards will be viewed as important and will be a successful "way of doing software."
2. The requirements of society and profession will be met.
3. Consensus standards will be looked upon more positively in the market place.
4. Standards will be perceived as an aid rather than an impediment to software development.

Additionally it is based on the idea that the work of project P1002 meets its goals in that it provides a framework within which software engineering standards can be developed and used in order to guide the construction of software that will satisfy the increasingly complicated and complex use of computers in present and future society.

In order to show the utility of the work of the P1002 project the presentation of this paper shows the application of the draft standard taxonomy in classifying already existing standards and potential standards. A collection of public standards available through trade associations, government agencies, national societies other than IEEE is classified by means of the taxonomy framework. In a separate application the IEEE software engineering standards which have been already developed and have passed consensus approval as well as those potential standards under development as IEEE-CS SESS projects are classified by means of the draft taxonomy. With this classification the software engineering standards activity within IEEE is evaluated using the taxonomy framework.

Appreciation is expressed to the P1002 working group and its product without which this paper could not be written or presented. In particular appreciation is expressed to P1002 working group chairman, Leonard Tripp, for his work in the classification examples shown. The Draft Standard Taxonomy for Software Engineering Standards is currently in ballot. Copies of the draft taxonomy are available from John W. Horch

M.S. 178

Cummings Research Park
Teledyne Brown Engineering
Huntsville, AL 35807

Members and Affiliate Members of IEEE are encouraged to evaluate and vote on the draft standard that is the work of the P1002 project.

SAGE
THE CLEMSON UNIVERSITY SYSTOLIC ARRAY GENERATING ENGINE

Roy P. Pargas and Keith R. Allen
Department of Computer Science
Clemson University, Clemson, SC 29634-1906

ABSTRACT

This paper describes the Clemson University Systolic Array Generating Engine (SAGE). SAGE allows a user to key in a nested loop algorithm in a high-level language and, for certain types of algorithms, such as convolution, matrix-vector multiplication, matrix-matrix multiplication, polynomial evaluation, and linear recurrences, automatically generates systolic array solutions. The paper concludes by listing future plans for SAGE, and some thoughts on possible directions in the automatic generation of systolic arrays.

INTRODUCTION

Designing systolic arrays to implement algorithms has traditionally been a time-consuming process. The problem does not lie in algorithm development; the algorithm being implemented at the time a systolic array is designed is typically well-understood. Rather, the major portion of systolic array development time is devoted to the often-tedious process of mapping data flow in the algorithm onto a regularly-interconnected array of computational cells. To be successful, the designer must accomplish this mapping in such a way that the data values associated with the computation enter and move through array cells in a very orderly (i.e. "systolic") fashion.

Until recently, designing systolic arrays has been approached in an ad hoc, seat-of-the-pants manner. Research in the past four years has shown, however, that it is possible to substantially automate the process of designing systolic arrays, for certain classes of loop algorithms [1,2,3,4,5,6,7,8]. Lam and Mostow[2] describe an experimental prototype program, dubbed *Sys*, which generates systolic designs by applying a series of transformations to a source algorithm which has been annotated with advice as to how to allocate operations to hardware cells. The authors argue that their transformational method has advantages for handling complex algorithms, and for verifying correctness of designs obtained.

In a paper by Li and Wah[3], the problem of designing a one- or two-dimensional systolic array for computing certain recurrences is formulated as a linear programming problem. Characteristics of systolic arrays are parameterized by data flow velocities, spatial distribution, and periods of computation. Constraint equations are then functions of these parameters, and linear programming is employed to derive optimal designs for various objective functions of computation period and number of processing cells.

A technique for generating systolic designs from a "space-time representation" for a recurrence algorithm is described by Steiglitz and Capello[7]. First the algorithm is imbedded in N-dimensional space in a graph which embodies dataflow dependencies of the algorithm, and it is imagined that the entire algorithm is computed in parallel, at many points in space, at a single instant of time. To generate various systolic designs for the algorithm, the space-time representation is typically operated upon by an affine transformation, and a spatial dimension of the image is "traded" for the time dimension.

Hornick[8] describes a class of transformations on systolic networks which alter network topology while preserving timing of its computations. Such transformations are used to demonstrate the equivalence of existing designs, or to obtain new designs from an existing design.

The basic method underlying SAGE is based on work first done by Moldovan[4,5]. Allen and Pargas[1] undertook a case study of this method and showed how to use the method to generate various designs for convolution and matrix multiplication. The same method was extended by Moldovan and Fortes[6] to deal with the problem of partitioning a large nested loop algorithm and mapping it onto a small systolic array, in such a way that accuracy of results is not compromised, and communication overhead between subproblems is minimized. They developed a software package called ADVIS to aid in implementing their partitioning/mapping process.

BRIEF OVERVIEW OF SAGE

SAGE (Version 1.0) is written in Pascal and runs on an IBM-PC. The program allows the user to key in a nested loop algorithm in a high-level language, and first analyzes the source loop algorithm to show dependences among its variables. If analysis of these dependences shows that a systolic solution is feasible, the user may proceed to ask for one or more solutions to be generated.

The solution space of systolic designs for a given loop algorithm is usually infinite, so SAGE provides the user with various means for guiding the solution search so as to produce only those solutions which are "reasonable", or "interesting". A typical criterion governing the user's interest may be, for example, the way in which the systolic computation interfaces with the larger computation in which it is imbedded. Once a solution of interest is generated, SAGE allows the user to see finer details of the systolic design by single-stepping through the computation and displaying cell locations of variables at each time step.

The loop algorithms must satisfy certain requirements:

1. They must be nested loops, two or three deep.
2. Only the innermost loop may contain an assignment statement.
3. Only one assignment statement is allowed.

Although the requirements appear to be somewhat restrictive, we have found that several different types of loop algorithms satisfy these requirements. These include loop algorithms for convolution, matrix-matrix multiplication, matrix-vector multiplication, polynomial evaluation, and the solution of recurrences. For each algorithm, multiple solutions have been generated.

Briefly, SAGE searches for a systolic array solution by performing three steps.

1. Determine the dependence vector, X , for each of the variables used in the loop algorithm.
2. If the dependence vector X is constant, find a transformation matrix T such that $TX=V$ and the first component of the image vector V (i.e., $V[1]$) is positive. The reason for this is that the first component of V is interpreted as the "time" component of the systolic array. The restriction, in effect, requires that time only go forward.
3. Build the systolic array from information gained by analyzing the vectors V , and by applying T to the loop indices of the source algorithm.

SAGE is entirely menu-driven. Its main menu includes the following user options.

1. Collect loop information. The user enters information such as loop limits and array variable names and indices, as given by the source loop algorithm. As soon as the information is complete, SAGE determines whether dependences among the variables are constant, assuring a systolic array solution. If a non-constant dependence is found, the user may still be able to develop a solution, although one is not guaranteed.
2. Display loop information. Dependence vectors may be inspected by the user.
3. Matrix selection. After the user specifies upper and lower limits on the values of the transformation matrices, SAGE conducts a search. Each suitable matrix found is displayed together with information on the systolic array that it generates. The information includes the number of systolic cells used and the number of time steps required by the systolic array to execute the algorithm. The user has the option of continuing the search, forward or backward, or of selecting a specific matrix for tracing.
4. Tracing systolic array execution. The user may single-step through the systolic array solution, either forward or backward. The variables are displayed moving from cell to cell. The user may select which variables to display, and which to temporarily mask out, allowing the user to focus on the movement of only selected variables.

The interested reader is referred to reports by Allen and Pargas[1,9] in which SAGE is described in greater detail.

CONCLUSION AND FUTURE WORK

Initial experience with SAGE indicates that, for appropriate source algorithms, the search for a systolic array solution becomes almost trivial. The user no longer faces the task of tracing the movement of the data through the cells. All of the tedious work is done by the program. We continue to experiment with SAGE. In particular, we continue to search for other problems whose algorithms satisfy, or may be modified to satisfy, the requirements described above.

Work also continues on SAGE itself. Future versions will incorporate at least the following features.

1. Multiple assignment statements in the innermost loop.

2. Greater user control over characteristics of the transformation matrices returned by SAGE. The user will be able to specify, for example, that SAGE return only systolic solutions in which a specific variable remains fixed in the cells, or that a variable moves right to left, or that a variable moves at a specified speed.
3. Equivalence classes of solutions. Hornick[4] describes a method for determining a family of systolic solutions to a problem given one solution. Future versions of SAGE will provide the user with the option of searching for (or ignoring) members of the equivalence class of a known and specified solution.

Finally, research in the general area of automatic generation of systolic arrays continues at Clemson University. As experience with the program grows, we plan to develop a design methodology for the construction of systolic arrays. The goal will be a general and systematic method for the generation of systolic arrays for a wide variety of loop algorithms.

REFERENCES

1. K.R. ALLEN and R.P. PARGAS. On compiling loop algorithms onto systolic arrays. Second SIAM Conference on Parallel Processing for Scientific Computing, (November 18-21, 1985), Norfolk, VA.
2. M.S. LAM and J. MOSTOW. A transformational model of VLSI systolic design. Computer, (February 1985), 42-52.
3. G. LI and B. WAH. The design of optimal systolic arrays. IEEE Transactions on Computers C-34, (January 1985), 66-77.
4. D.I. MOLDOVAN. On the analysis and synthesis of VLSI algorithms. IEEE Transactions on Computers C-31, (November 1982), 1121-1126.
5. D.I. MOLDOVAN. On the design of algorithms for VLSI systolic arrays. Proceedings of the IEEE 71, (January 1983), 113-120.
6. D.I. MOLDOVAN and J.A.B. FORTES. Partitioning and mapping algorithms into fixed-size systolic arrays. IEEE Transactions on Computers C-35, (January 1986), 1-12.
7. K. STEIGLITZ and P.R. CAPELLO. Unifying VLSI Array Design with Linear Transformations of Space-Time. Advances in Computing Research, vol. 2, VLSI Theory, (F.P. Preparata, Ed.), 1984, 23-65.
8. S.W. HORNICK. A unified approach to the analysis and synthesis of systolic arrays. Masters thesis, Department of Electrical Engineering, University of Illinois at Urbana-Champaign, 1985.
9. R.P. PARGAS. SAGE: Clemson University Systolic Array Generating Engine. User Guide, Version 1.0. Technical Report #86-4-9, Department of Computer Science, Clemson University, Clemson, SC 29634-1906. April, 1986.

John W. Stoughton and Ronald B. Mize Jr.
Department of Electrical Engineering
Old Dominion University
Norfolk, VA 23508

The data flow model of computation has spawned diverse approaches to the strategies associated with concurrent processing. In particular, the distributed computer system (DCS) is an important class of architecture as a strategy for concurrent processing. Problems associated with the DCS concept include the decentralized control and task-to-resource assignment. The problems stem from the forward flow of both data and control/assignment information or tokens. Scheduling of the data flow becomes difficult or awkward as algorithm information is decomposed to meet a particular architecture strategy. Hence, the imperfect match up between actual architecture data flow and desirable algorithm data flow leads to difficulty in the analysis of the execution behavior of the decomposed algorithm in the concurrent processing environment.

Node Marked Graph Model

1. Functional units are processors with local memory for program storage and temporary input and output data containers.
2. The data memory is global to all functional units. The inputs associated with each computational graph node (process) correspond to fixed data containers in the global data memory.
3. The computational process cannot be assigned to a processor (functional unit) unless a Functional unit is available and all inputs to the computation are available.

[illegible]

reproduction on NMIs shown in Figure 1. The peak at 10.1% of the adult population

and nodes (transitions) are presented below.

7710 2112 = 3212 = 2.

```

I1-f      Input-1 buffer full (data available)
I1-e      Input-1 buffer empty (status)
I2-f      Input-2 buffer full (data available)
I2-e      Input-2 buffer empty (status)
DR        Data ready in functional unit
NB        Process not busy
PR        Process or computation is complete

```

NMG edge labels (cont'd)

01-e	Output-1 buffer empty (status)
01-f	Output-1 buffer full (data available)
02-e	Output-2 buffer empty (status)
02-f	Output-2 buffer full (data available)

NML nodes.

- A. Transition A fires when I1,I2-f, and NB tokens are available. Implicit in the firing rule is priority assignment and availability of a functional resource to assign to the particular task.
- B. Transition B fires when DR token is available. This transition corresponds to the actual computation event.
- C. Transition C fires when O1,O2-e, and PR tokens are available.

The computational marked graph (CMG) of a particular algorithm is composed of the interconnection of NMGs by joining the various input arcs to the corresponding output arcs of predecessor NMGs. Data flow and buffer-full conditions are represented by directed arcs corresponding to the directed data flow of the original computational flow graph. However, markings associated with status and control are described on directed arcs in the direction opposite to the data flow. This observation leads to a consistent view of the data flow and rules to be followed by any data driven architecture to implement the process. The algorithm graph and computational marked graph are described in more detail and are illustrated in Figures 2 and 3, respectively.

Data Driven Architecture

The architecture must allow the execution of the CMG whose nodes are enabled by data and fired by the availability of a functional unit. A multiple resource architecture is shown in Figure 4. The architecture is a data-driven structure which is a natural consequence of meeting the requirements of the Petri-net model.

The candidate architecture consists of four different units and two interconnecting buses. The hardware units consist of k-functional units (FU), one high speed data memory (DM), one (or more) I/O units and a token manager (TM). Control and status information are passed between the FUs and the TM, the token bus. Data packets are passed to and from the various FUs and the DM by way of the data bus. The I/O units obey all protocol rules of the TM and pass out a flow data to be entered or removed from the network. The candidate architecture is shown in Figure 1. The candidate architecture is based on the following assumptions:

- 1. The network is a closed system with a fixed number of nodes.
- 2. The network is a single hop network with a fixed number of nodes.
- 3. The network is a single hop network with a fixed number of nodes.
- 4. The network is a single hop network with a fixed number of nodes.
- 5. The network is a single hop network with a fixed number of nodes.

The candidate architecture is based on the following assumptions:

- 1. The network is a closed system with a fixed number of nodes.
- 2. The network is a single hop network with a fixed number of nodes.
- 3. The network is a single hop network with a fixed number of nodes.
- 4. The network is a single hop network with a fixed number of nodes.
- 5. The network is a single hop network with a fixed number of nodes.

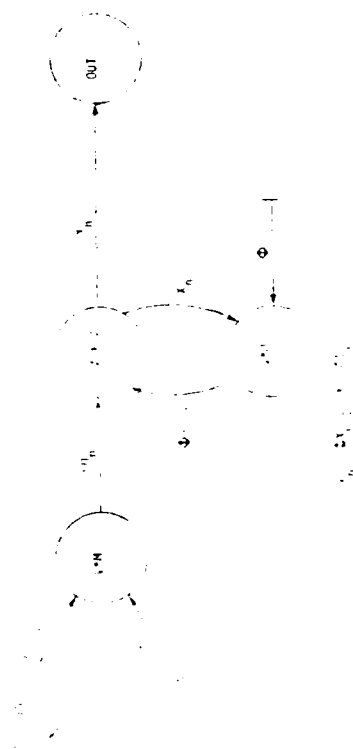


Figure 1. Node Marked Graph

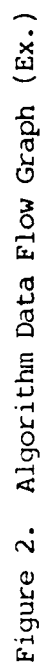


Figure 2. Algorithm Data Flow Graph (Ex.)

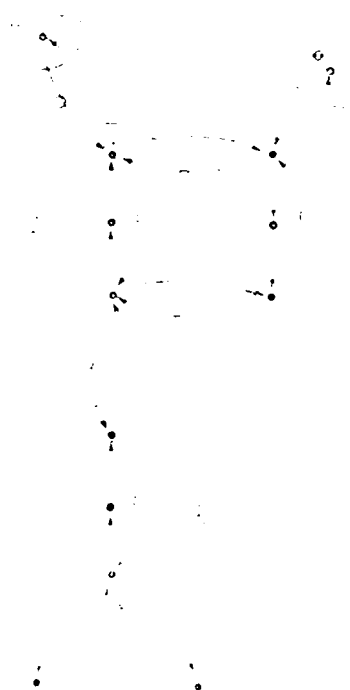


Figure 3. Computational Marked Graph (Example)

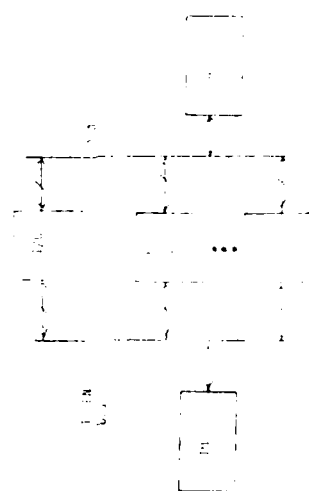


Figure 4. Candidate Architecture

Portable (and disposable) Interpreters

François-Xavier TESTARD-VAILLANT

Ecole Normale Supérieure
Avenue de la grille d'honneur 92211 Saint Cloud
FRANCE
&
U.P.M.C.
Laboratoire d'Informatique Théorique et Programmation
4, Place Jussieu 75252 Paris Cedex 05
FRANCE

Summary

This paper proposes a methodology for quick and not dirty interpreter implementations in high level languages. It is based upon a formal and operational description of stack manipulations using the mechanism of exceptions. This formalism provides a powerful tool to write or extend an interpreter from its meta-circular definition - e.g. Lisp or Prolog.

Keywords: Portable software, Exceptions, Interpreter, Lisp, Prolog.

1. Introduction

The exception mechanism is present in many programming languages. The conditions of PL/1 [PL/178] has been the first but not the least. Clu [Liskov79] and Ada [Ansi83] own less powerful statements of that type. Recently the problem of its implementation and use in Modula-2 [Hopkin86] and Pascal has raised. Common-LISP [Steele 84] does not (presently) include this feature. Note that the three last languages are powerfull enough to allow simulation [Testard-Vaillant86].

1.1. Exceptional situations

We call **exceptional** those situations in which the interpreter has to walk down the recursion stack - i.e. those cases when the interpretive process follows no longer a strict stack discipline.

There are basically three of them :

- 1) The **Garbage Collector**, which has to take into account not only past bindings (shallow) but also past values in the recursion stack.
- 2) The **non-local exit mechanism**, which is clearly exceptional. Note that the whole of error handling can be brought under this heading.
- 3) The **tail-recursive situations checking** [Greussay77, Saint-James84], in dynamically-scoped LISP interpreters which requires a very precise examination of the stack to a depth unknown in advance.

Now, these three points cannot be easily treated in isolation. Actually, in current practice of interpreter writing, they pervade virtually the whole code : every single module has to beware of some aspects of "*the gang of three*".

Our *gang of three* is not absolutely necessary but A.I. applications often spend lavishly both run-time and memory-space. The salient point of the third part is that it ensures the user that the memory-space devoted to the recursion stack will be as small as possible. This encourages the use of recursion which becomes an efficient tool for programming (from a **run-time** point of view) and allows extension of LISP toward **multi-processing**.

1.2. Connections between exceptions and exceptional situations

Generally speaking, one could say that a LISP system is made up of two parts: The **normal** one, which is usually specified by some metacircular model [Rivières84] and the **exceptional** one, for which no formal description is given.

Unfortunately, the **normal** and **exceptional** parts of every module are treated together, in an all but clearly organized way. As a consequence of this state of things, some unpleasant features are commonly accepted such as the destruction of the stack when a *thrown* value doesn't find its *catch* [Chailloux84].

All the sequencing functions (*progn*, *or*, and ...) have to take precautions i.e. to free the top of the stack to allow the tail-recursion checking [Saint-James84]. This tangled way of writing practically prevents the verification that the **normal** part is indeed faithful to the metacircular definition - not to mention inquiries regarding the **exceptional** part.

We argue that the present situation is largely due to the illusion that **exceptional** features can be treated in an off-hand way, by immoderate use of the facilities permitted by assembly language programming, which does not encourage a systematic point of view. On the contrary, we adopt the attitude that interpreter writing, as the rest of systems programming, should be performed in high level languages. Furthermore we claim that high level languages must be really used as high level language i.e. we will entrust them the management of the recursion stack. This clearly necessitates the formalization of stack manipulations.

We propose the notion of an **exception** as the adequate tool for our stack-perusing tasks.

- 1- The Garbage Collector deals with the *gc* exception raised by *cons*.
- 2- The dynamic non local exits deal with the *escape* exception raised by *throw*.
- 3- The tail-recursive situations checking deals with the *tail* exception raised by *apply*.

1.3. Overview of the interpreter

A LISP interpreter is a set of procedures. These procedures are twofold: the body directly translated from the metacircular model and the handlers which have to handle the *exceptional situations*. Though the specification of each *exceptional situation* will afford:

- 1- the name of the specialized exception,
- 2- the name of the procedure which initially raise the exception and
- 3- the kinds of handlers the set of procedures needs to handle this peculiar *exceptional situations*.

2. Exceptions behavior

Let us draw a parallel between exception raising and message passing in actor-based languages: the procedure which needs to modify the stack or only to get information from it will send a message to the closest (calling) procedure precisely by raising an exception. This mechanism does not have to destroy the stack except if it is explicitly asked. If the receiving procedure is not able to understand (handle) the message it will be forwarded down to the closest one in the recursion stack which will act as the *proxy* in ACT1 terminology. On the contrary if the receiving procedure owns a method for this message (a handler for this exception) it can either kill the current message (exception), propagate it i.e. transmit it to its own calling procedure or return it to the sender.

3. Fixing the stack problem

3.1. The Garbage Collector

Let's take a widely used Garbage Collection algorithm - e.g. the *Mark and Sweep* one.

The Garbage Collector is called by cons. It raises the exception called gc. There are two kinds of handlers: those which belong to a procedure which used cons cells for its owns and others. The first kind have to mark the said cons cells. Both of them have to propagate the gc exception.

3.2. Non-local exit

The non-local exit is the work of throw. It raises the exception called escape. There are three kinds of handlers: thoses which only propagate the exception, the one which unbind what needs to be before to propagate the exception and the one which kill the exception.

Apply or more precisely the specialized part of it which deals with λ -expressions owns a second kind of handler and catch-all clearly owns a third kind of handler.

3.3. Iterative interpretation of tail-recursive calls

This is a well-known problem both in lexical binding context [Steele76a, Steele76b, Steele77] and in dynamic binding context [Saint-James84].

The tail-recursive situation checking is the work of apply. It raises the exception called tail. There are three kinds of handlers: those which return the exception meaning the current situation is not tail-recursive, those which propagate the exception meaning the current situation may be tail-recursive and that which can kill the exception.

The handler of cons belongs to the first kind whereas the progn's one checks if the expression that it is evaluating, is the last one to decide to return or to propagate. The part of apply specialized in λ -expressions (apply_expr) kills the exception after the updating of the environment if the current call is identical to its.

Improvements

We will show that the crossed tail-recursive call is nothing but the addition of a new exception with the description of the raising function and the kinds of handlers needed.

The tail-recursive situation checking is the work of the `apply_expr`. It raises the exception called `crossed`. There are three kinds of handlers which are basically the same as the tail exception except for the third kind. Rather than kill the exception when a crossed tail-recursion has been discovered the handler has to update the environment and then to return it. Coming back to `apply_expr` the exception will be killed after the updating of the current λ -expression's body.

More complex cases such as left associative envelope - e.g. `+` and `*`, are described in the same way.

4. General view of portable (and disposable) interpreters

The exception mechanism allows the formal and operational description of stack manipulations. This ends up to quick and not dirty implementations of LISP interpreters in high level languages. Following P. Greussay [Greussay77], we can now assert that the quicker way to implement an efficient Prolog interpreter (for example) is not to write it in LISP but to deduce it from a LISP interpreter clearly specified.

Acknowledgments

The author wishes to thank Jean-François Perrot and E. Neidl for their encouragements during this work.

References

[Ansi83]

Reference Manual for the Ada Programming Language (ANSI/MIL-STD 1815 A), Alcy, La Celle-Saint-Cloud, January 1983.

[Burke85]

Burke, G.S., *Catching Common Lisp Forum -- Mid* <MIT-MC.ARPA.724785.851119.GSB>, November 1985.

[Chailloux84]

Chailloux, J., Devin, M. and Hullot, J.M., *LeLisp, a Portable and Efficient Lisp System*, Fifth ACM Conference on Lisp and Functional Programming, Austin Texas, August 1984.

[Greussay77]

Greussay, P., *Contribution à la définition interprétative et à l'implémentation des lambda-langages*, Thèse d'état, Université PARIS VII, PARIS, Novembre 1977.

[Hofkin86]

Hofkin B., *Exceptions in Modula-2*, UUCP News Mid <8602032056.AA26124@calmasd.CALMA.UUCP>, February 1986.

[Liskov79]

Liskov, B., *CLU Reference Manual*, M.I.T., Cambridge, October 1979.

[Moon75]

Moon, D.A., *MacLISP reference Manual*, M.I.T., Cambridge, 1975.

[Moon85]

Moon, D.A., *Universal Catch*, Common Lisp Forum -- Mid <851120172629.1.MOON@EUPHRATES.SRC.Symbolics.COM>, November

1985.

[Saint-James84]

Saint-James, E., *Recursion is more efficient than iteration*, Fifth ACM Conference on Lisp and Functional Programming, Austin Texas, August 1984.

[Saint-James86]

Saint-James, E., *Echappements et pas à pas*, LITP (rapport interne à paraître), Paris, 1986.

[Smith84]

Smith, B.C. and des Rivières, J., *The Implementation of Procedurally Reflexive Languages*, Fifth ACM Conference on Lisp and Functional Programming, Austin Texas, August 1984.

[Steele76a]

Steele, G.L., *Lambda: The ultimate declarative*, AI memo 379 MIT, Cambridge Mass., 1976.

[Steele76b]

Steele, G.L. and Sussman G.J., *Lambda: The ultimate imperative*, AI memo 353 MIT, Cambridge Mass, 1976.

[Steele77]

Steele, G.L., *Debunking the expensive procedure call myth*, Proc. of the ACM conference, 1977.

[Steele84]

Steele, G.L., *COMMON LISP (The language)*, Digital Press (DEC), 1984.

[Testard-Vaillant85]

Testard-Vaillant, F.-X., *Interprétation, en langage évolué, de langages très évolués*, Thèse de troisième cycle, U.P.M.C., Paris, December 1985.

[Testard-Vaillant86]

Testard-Vaillant, F.-X.,
Implementing Exceptions, à
paraître, 1986.

[White79]

A White, J., *NIL: A
perspective*, Proc. of the
Macsyma Users Conference,
Washington D.C., June 1979.

[Wirth80]

Wirth, N., *Modula-2*, Rap-
port numero 36, ETH Zurich,
March 80.

**Session 14C: Logic and Functional
Programming**

Chairperson: Jon Mauney
North Carolina State University

A RULE-BASED LISP DIALECT TRANSLATOR USING PARAMODULATION

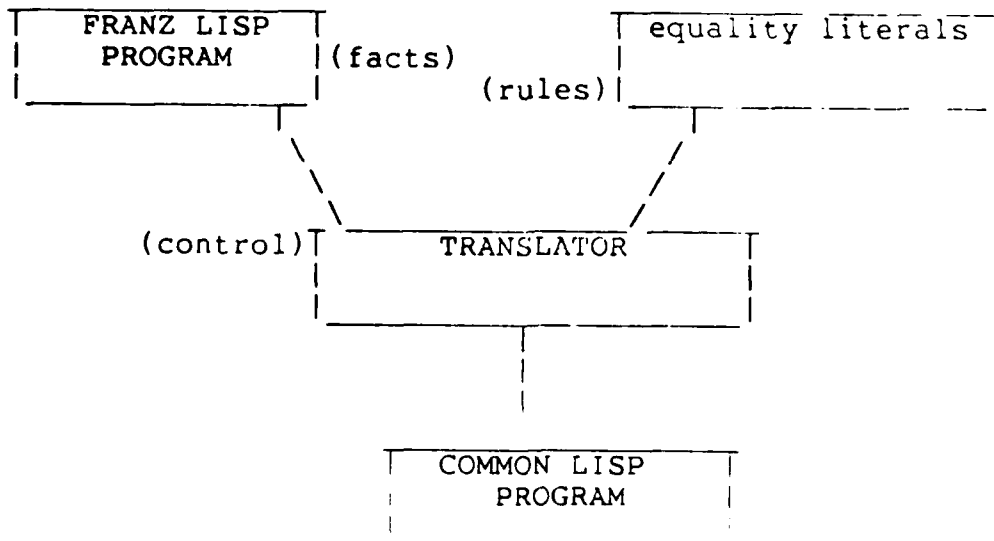
Michael Dowell
Yoshiyasu Takefuji

Center for Machine Intelligence
Department of Electrical and Computer Engineering
University of South Carolina
Columbia, SC 29208
Phone: (803) 777-5099

ABSTRACT

In this paper a rule-based Lisp dialect translator (LDT) is presented. The advantage of using a rule-based system is to allow the user to supply his own rules for translation and thus the translator can be considered a general purpose dialect converter. The translation being used for development is Franz to Common Lisp.

1 INTRODUCTION



The LDT takes advantage of the fact that the structure and many of the semantics of different dialects of Lisp are the same and therefore do not need to be changed. The LDT makes both one-to-many and many-to-many translations. The rule form, the different types of translation and the relationship between Franz and Common Lisp will be discussed in the following sections.

2 RULE FORM

The rules are in a paramodulation form, as an example of the one-to-one substitution:

The relationship: EQUAL (add : +)

The Franz function: (defun example_1 () (add 2 3))

The Common function: (defun example_1 () (+ 2 3))

Paramodulation occurred from the "add" term into the "+" term. The LDT shall restrict the form of the equality literal such that the from term is always to be the left-hand side of the equality literal and the into term is always to be the right-hand side of the equality literal.

3 TRANSFORMATIONS

3.1 One-to-one Transformations

The rule form for one-to-one transformations is the simplest to derive with the new_expression directly replacing the old_expression. See the previous example.

3.2 One-to-many Transformations

The rule form for one-to-many transformations involves matching. This is done by using variables within the expressions. The variables in different equality literals are completely independent, even if they have the same name. The first part of the rule will allow for single-atom instantiation or multiple-atom instantiation. Single-atom instantiation will be specified by using the "?" symbol.

For example: match '(? x) 'hello

will cause x to be instantiated to "hello".

Multiple-atom instantiation will be specified by using the "+" symbol.

For example: match '(+ x) '(hello world)

will cause x to be instantiated to '(hello world).

The second part of the rule will have the variables replaced by the instantiated value, regardless of either single- or multiple-atom instantiation.

As an example of one-to-many substitution:

The relationship: `EQUAL (nequal (? x) (? y) : not (equal x y))`

The Franz function: `(defun example_2 ()
 (cond ((nequal 'a 'b) (print 'a))))`

The Common function: `(defun example_2 ()
 (cond ((not (equal 'a 'b)) (print 'a))))`

3.3 Many-to-many Transformations

This type of translation can be seen as a combination of one-to-one and one-to-many translation. With operations being performed on the variables and a direct substitution between the two instructions.

As an example of many-to-many substitution:
The relationship :

`EQUAL (append1 (? x) (? y) : append x (list y))`

The Franz function:

`(defun example_3 () (append1 '(a b c) 'a))`

The Common function :

`(defun example_3 () (append '(a b c) (list 'a)))`

3.4 Special One-to-many Transformation

To translate superparentheses from Franz to Common Lisp a special type of translation is needed. In Franz Lisp a right superparenthesis is represented by "]" and can close off as many open left parentheses as needed until the end of the function is reached or until an open left superparenthesis is encountered. A left superparenthesis is represented by "[" and closes one right superparenthesis. Superparentheses are not allowed in Common Lisp so a transformation is needed to change superparenthesis to parenthesis. At present this substitution takes place in a function and is not implemented with a rule.

The Franz function: `(defun example_4 ()
 (cond [null ()] (print "hello")])`

The Common function: `defun example_4 ()
 (cond [null ()] (print "hello"))`

Semi-Applicative Programming

N.S. Sridharan

BBN Laboratories Inc., 10 Moulton St. Cambridge, MA 02238

Arpanet: Sridharan@C BBN:ARPA

Novel parallel machines are being designed and built. Amid the loud applause for the ingenuity of the ideas and implications of their success, we also hear the remark "But how are you going to program such a beast?", thereby implying that the software problem remains once the hardware is designed. Similarly, several attempts are being made at parallel programming language design, one hears the remark "What we really need are ways of thinking and problem solving that incorporate parallelism". This reaction stems from the view that a programming language is merely a notation and new developments in programming methodology are essential for the proper use of the next generation of computers.

We aim at developing a programming language and programming methodology that allow effective use of medium-scale, medium-grain parallelism, support correct program development, and allow effective, error-free control of program behavior through a variety of means.

As an initial set of problems to study in the project, we are investigating search algorithms (alpha-beta, branch and bound, backtrack), constraint propagation and marker propagation algorithms, constraint satisfaction and relaxation algorithms and parsing algorithms.

The full length version of this paper [9] presents the results of study on parsing algorithms for context-free grammars. We start with the specification of a recognizer for context-free grammars in Chomsky-Normal-Form (CNF) and derive by transformations a variety of different purely applicative parallel recognition algorithms. We then introduce program annotations and display semi-applicative algorithms. In one algorithm we indicate how adaptive scheduling can control the behavior exhibited by the algorithm. Thus we hope the reader observes the use of transformations, annotations and scheduling as means of controlling program behavior.

1 Resource control in applicative programs

We are experimenting with a programming language, SALT, that has an *implicitly* parallel applicative language as the base language, and then we introduce constructs

¹This research was sponsored by the Advanced Research Projects Agency of the Department of Defense and was monitored by ONR under Contract No. N00014-85-C-0079. The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

that inhibit parallelism. The base language, SALT, uses pure LISP [7] as a foundation and blends in interesting features of Prolog [6] and FP [1]. We introduce several techniques of controlling the behavior of functional programs without changing their meaning or functionality. (i) program annotation with constructs that have benign side-effects, (ii) program transformation, and (iii) adaptive scheduling. This combination yields us a *semi-applicative* programming language and an interesting programming methodology. Because the applicative language has simple clear parallel semantics, it facilitates doing program transformations. Because the expression of concurrency is implicit, this provides freedom in scheduling tasks and thus facilitates adaptive scheduling. *We believe that this approach combining implicit parallelism and explicit control will reduce the risk of introducing bugs in the process of tuning programs for performance*, as well as providing an antidote to Amdahl's law.

1.1 Program annotations

We have developed an initial design for an annotation language called PEPPER. Annotations in PEPPER include precedence control, function call/result caching [4, 5], and lazy (or demand-driven) evaluation. Our initial design document describes PEPPER annotations and gives several examples of their use. A key feature of PEPPER annotations is that their introduction will not alter the functional value of a program, and will affect only its run-time behavior.

1.2 Program transformation

Program transformation [2, 3, 8] of the source-level algorithms written in SALT is essential. There are two quite different reasons for considering program transformation. Firstly, given that PEPPER annotations are to be added to the text of a program, it is evident that *textually different but functionally equivalent programs offer different opportunities for adding annotations*, and hence provide different opportunities for achieving different behaviors. A programmer can make full use of annotations only in conjunction with the ability to do program transformations. Secondly, *different programs yield different data-dependency orderings*, thus allowing differing amounts of concurrency. Thus, program transformation is a technique for controlling the available concurrency in a program.

1.3 Adaptive scheduling

Adaptive scheduling can yield improved behavior with repeated runs of the same program. Adaptation requires monitoring and measuring run-time characteristics of SALT+PEPPER programs in order to make scheduling decisions dynamically. In working with virtual parallelism, the program only expresses 'available' concurrency without mandating what in fact will execute concurrently with what. The scheduler converts the available concurrency into physical concurrency, making its choices for running tasks by considering available resources, resource requirements and other attributes of tasks. The scheduler typically is a heuristic procedure and does not yield optimal behavior in all cases. Thus, another avenue open to the programmer is to tune some parameters of the scheduler to alter the behavior of the program. We feel this method of control is 'indirect', and while that is necessary, it is not likely to be sufficient. It may be useful to construct an adaptive scheduler that tunes its parameters based on empirical measurements.

2 Research Results: Recognition algorithms for CNF grammars

The initial specification is a standard mathematical definition for what a recognition algorithm should do. In several steps of transformation, we derive a variety of parallel recognizers (see Figure 1). In the literature there are two well-known parsing algorithms. Cocke, Younger and Kasami, independently reported on the development of a bottom-up parser that works on a restricted form of grammars called the Chomsky-Normal-Form. Earley published an algorithm that works with unrestricted context-free grammars and parses an input string in a top-down fashion using left-context to limit search. Most of us, including the author, have admired the cleverness of such algorithms. One of the things we do in the full paper is to demonstrate that these algorithms have real close affinities to each other and that they can be arrived at systematically. Similar derivations of various SORT algorithms have been previously presented in the literature. All of these have been successful only at deriving a few of the already known sort algorithms, none of these derive any new SORT algorithm. Can one readily adopt these transformational techniques to derive new algorithms? Our attempt hopefully convinces the reader that new parallel algorithms can be derived systematically.

Acknowledgement. It is my pleasure to thank Andy Haas for the help he has offered always with enthusiasm. He has also pushed me to search for simplicity for which I am grateful. I also thank my several colleagues at BBN Laboratories who have provided me their generous help and support.

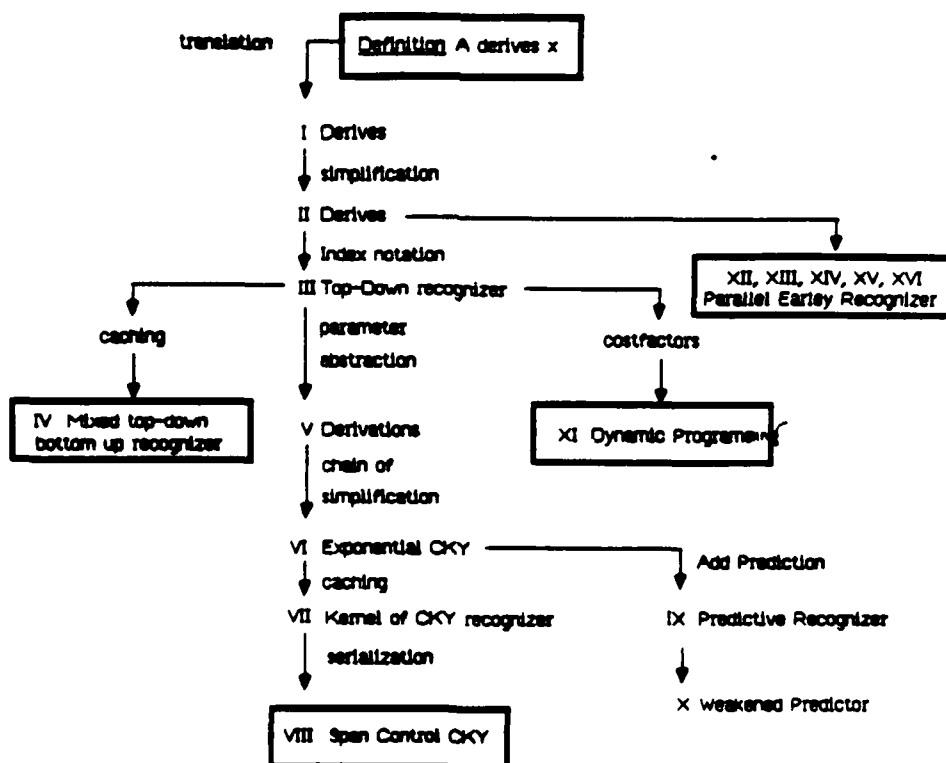


Figure 1. Chart showing transformational derivation of different algorithms for context-free recognition using Chomsky Normal Form grammars

References

- [1] J. Backus.
Can programming be liberated from the Von Neumann style?
Communications of the ACM 21(8) 613-641, July, 1982.
- [2] R. Balzer.
A 15 year perspective on automatic programming
IEEE Transaction on Software Engineering SE-11(11) 1257-68, November, 1985
- [3] W. Bibel.
Syntax-directed, semantics-supported program synthesis.
Artificial Intelligence 14.243-261, 1980.
- [4] R.S. Bird.
Tabulation techniques for recursive programs
Computing Surveys 12(4).403-417, December, 1980.
- [5] R.M. Keller and M. Ronan Sleep.
Applicative caching.
In *Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architecture*, pages 131-140 Association for Computing Machinery, October, 1981.
- [6] R. Kowalski.
Logic for Problem Solving.
Elsevier North-Holland, New York, 1979.
- [7] J. McCarthy et al.
LISP 1.5 Programmer's Manual.
MIT Press, Cambridge, MA, 1963.
- [8] H. Partsch and R. Steinbruggen
Program transformation systems
Computing Surveys 15(3).199-236, 1983
- [9] N.S. Sridharan.
Semi-Applicative Programming. Examples of context-free recognizers
Technical Report 6135, BBN Laboratories Inc., November, 1985

Experimenting with parallel programming in logic

by
Abraham Waksman
Temple University
Philadelphia, PA 19122

We present an extension to the logic programming language Prolog, SISPRO (Super 'is' Prolog). This extension is on a very small scale but is typical of other efforts in this area to extend the capability of purely logic programs to handle also functional constructs. The motivation for efforts in this direction is to remove obstacles in bringing about the further development of a very promising software tool.

The particular extension to Prolog to be discussed is realized by the introduction of a rewrite rule permitting the specification of functional terms to reside side by side with logical terms. A new local variable assignment operator is also introduced to take the place of the conventional 'is' operator. In addition to assignment to a local variable, results of simple arithmetic, SIS, the new operator, can also bind to the local variable results of function evaluation. In fact there is no restriction on the bindings capabilities of SIS.

The implementation of SISPRO was achieved by the creation of a small interpreter written in regular Prolog. This is but a small example of the use of Prolog as a language for the creation of runnable specification for software. The SISPRO interpreter will evaluate functions defined recursively and at the same time pass regular Prolog terms to the Prolog interpreter for conventional evaluation.

```
append([],L,L).  
append([X|L1],L2,[X|L3]):- append([L1,L2,L3]).
```

Example 1a. Standard Prolog does not require any specific variable binding order, thus achieving "multi directionality"

```
append([],B) => B.  
append([X|Y],B) => ([X| append(Y,B)]).
```

Example 1b. SISPRO requires that the input variables be bound at the beginning of execution, thus achieving "uni directionality"

As technology evolves, there is greater hope to free the software developer from the Von Neumann model of computation. Conventional or imperative programming languages are intimately tied to the Von Neumann model of computation. On the other hand, declarative languages such as Lisp and Prolog have their origin in mathematical formalism which have developed totally separate from any consideration of implementation by computation devices. This freedom resulted in formalisms which achieved greater expressive power, possibility for formal manipulation and ease of parallel evaluation. These attributes might very well influence the direction of future computer implementations.

Declarative languages have taken two forms: functional with lambda calculus and recursion equations systems, and logic programming based on procedural interpretation of the first order predicate calculus.

Logic programs are very amenable to parallel execution. We can view the computation of a logic program as a process of constructing a proof to the goal statement based on the axioms available to the program. The search space consisting of these axioms is often described as an And/Or tree, where an And node correspond to a conjunctive goal and an Or node correspond to the different ways a unit goal can be reduced.

The Prolog interpreter selects the first listed Or term and then proceeds to travers the And nodes from left to right. Or parallelism occurs when several Or paths are tried at once. And parallelism occurs when all the conjunctive pathes are tried at once. Goals in the conjunctive nodes may have variables in common and therefore are not independent of each other. Thus the coordination of the And parallel execution is necessary.

To date, the fundamental approach to the parallel evaluation of Prolog clauses follows the pattern of constructing experssions of the following form:

$A:- G_1, G_2, \dots, G_m \mid B_1, B_2, \dots, B_n. \quad m, n \geq 0$

G's and B's are atomic formulas (unit goals) and A is a clause head. The G's are the guards and the B's are the body. A variable within an atomic formula in the body of a clause can assume an independent role when evaluated during parallel execution, or it can assume a dependent role, resulting in suspension of execution for the term until a value is "communicated" to it from a term with the same variable in an independent role.

The essential difference between functional and logic programs is mostly considered in their "input output directionality". Functional languages are directional in that their programs make an explicit commitment about which quantities are inputs and which are outputs. Logic programs do not make such commitments. Non directionality, although endowing logic programs with greater expressibility, results in problems of efficiency of execution and problems in their parallel implementations.

The "input output directionality" disparity between functional and logic programs is the direct consequence of the calling mechanisms employed by the two languages. Functional languages employ 'pattern matching' while logic programs employ 'unification'. Unification subsumes pattern matching, herein lies the capabilities of logic programs not possessed by functional programs. In unification, both the goal and the head of the clause being resolved against are allowed to contain variables, and a successful unification produces two separete substitutions, Input substitution and output substitution. In pattern matching, only the head of the clause or the goal are allowed to contain variables, thus input or output directionality is implicitly specified permitting only input or only output substitution at runtime.

The objective strived for in the design of SISPRO was to introduce directionality via the specification of functional terms, for the purpose of execution efficiency. Evaluation in such cases is purely deterministic and backtracking is not needed by the interpreter for undoing previous bindings. The dual directionality of the logic component of the language remains unaffected and the programmer is free to switch back and forth between functional clauses and relational clauses.

In most parallel implementation of logic programs, in order to avoid ambiguity during run time assignment process, modality has to be specified and variables that appear in more than one formula have to be tagged. Modality refers to the specification of the mode of evaluation for the clause. Tagging refers to the identification of variables in terms of their role in the run time binding priorities. Such tagging goes by the names of: Read only variables, lazy or eager evaluation variables and consumers and producers variables.

In SISPRO delayed (lazy) evaluation of the second term of a rewrite rule, often occurs. When a variable is unified with a term of the form $\Rightarrow f(\text{foo})$, rewriting must not always be completed. Often, it is sufficient just to do normal Prolog unification to the term and proceed with the rest of the computation for a while. This is a form of delayed evaluation providing automatic moding in a parallel evaluation environment.

The runtime behavior of functional programs is much simpler to control than that of logic programs, particularly in parallel execution context. By reducing, whenever possible, the logic program to its functional equivalence, we achieve greater runtime efficiency, greater clarity in program statements and greater control over program execution.

```
qsort([]) => [].
qsort([X|U]) => append(qsort(U1),[X|qsort(U2)]):-
                    partition(X,U,U1,U2).
partition(X,[],[],[]).
partition(X,[Y|U],[Y|U1],U2):- Y <= X, partition(X,U,U1,U2).
partition(X,[Y|U],U1,[Y|U2]):- X < Y, partition(X,U,U1,U2).
```

Example 2. Definition of the sorting algorithm QuickSort as an hybrid of functional and logical expressions, resulting in a much more efficient parallel evaluation.

Concluding remarks:

The use of logic programming as runnable specifications of software systems is growing. In particular, its use in specifying parallel processes. For efforts in this direction to have an impact, logic programming needs to be extended in a number of directions. This note examined via an implementation example, one such direction for extension.

Parallel Architectures for Logic Programming¹

Vipin Kumar & Yow-Jian Lin
Artificial Intelligence Laboratory
Computer Science Department
University of Texas at Austin
Austin, Texas 78712

Summary

This project is investigating parallel logic programming architectures for AI applications. Logic programming provides a convenient paradigm for expressing parallelism because it allows the separation of logic and control in an algorithm. Potential parallelism in a logic program can be extremely large. A given sequence of subgoals B_1, \dots, B_n can be solved by solving each subgoal B_i in parallel (AND-parallelism). Solution of a subgoal B_i can be attempted by trying all possible rules in parallel whose heads match B_i (OR-parallelism). But this scheme generates extremely large number of concurrent activities, and can only be implemented on an idealized architecture having an unlimited number of processors and zero communication overhead. Furthermore, much of the work done by this unconstrained search may be redundant. For example, if two subgoals, B_1 and B_2 , share variables, then many incompatible solutions of B_1 and B_2 may be generated. If only one solution of the original goal is needed, then many solutions generated for a subgoal can be redundant. Since a practical parallel processor can only have finite amount of resources, we need to develop a strategy which creates parallel activities in a judicious manner.

In our research we are investigating various (constrained) ways of using parallelism in logic programs and architectures for implementing them. We have developed a parallel execution model for logic programming. The model is process based and uncovers both AND and OR parallelisms, which makes it particularly suited for AI applications. We have developed a strategy which controls the exploitation of AND-parallelism with the help of domain-specific information provided by the programmer, and has minimal run time overhead [4]. When subgoals share variables, failure of a subgoal may mean that some other subgoal needs to backtrack and generate a new solution. This can be done easily (by backtracking to the subgoal to the left of the failed subgoal) if the subgoals are solved strictly sequentially. But when AND-parallelism is being exploited, the backtracking decision becomes nontrivial. We have developed an algorithm for backtracking which is more efficient than the ones previously proposed [2], [1], and requires minimal run time overhead [6]. Another feature of our execution model is that it can be implemented in a fully distributed manner; i.e., the AND process solving a sequence of subgoals does not have to work as a message center for "child" OR processes [5].

Due to limited resources, we have to choose the relative degrees of permitted AND and OR parallelisms. The relative utility of these two parallelisms is greatly dependent

¹ This work was supported by Army Research Office grant #DAAG29-84-K-0060 to the Artificial Intelligence Laboratory at the

upon the specific logic program, as in some cases (e.g., for deterministic logic programs) OR-parallelism would only create redundant work, whereas in others (requiring lot of backtracking) OR-parallelism would be very useful. We plan to run simulations to study the trade off between AND and OR parallelisms for different kinds of logic programs. A study to investigate the effectiveness of various strategies for exploiting OR parallelism in logic programs is already underway [3].

Allocation of processors to processes in the physical architecture is another important issue needed to be solved. On one hand, since processes have to communicate, we do not want to allocate related processes far away in the network; on the other hand, since speedup is the main concern of parallel execution, we would like to distribute processes to every available processor. The trade-off here depends on the connectivity of underlying architecture. Different strategies for sharing data structures among processes can also be used to control the amount of message traffic generated. We are currently looking for some potential solutions to these problems and plan to test their usefulness via simulations.

REFERENCES

- [1] J.-H. Chang and A.M. Despain, Semi-Intelligent Backtracking of Prolog Based on a Static Data Dependency Analysis, *Proceedings of IEEE Symposium on Logic Programming*, pp. 10-21, August, 1985.
- [2] J.S. Conery and D.F. Kibler, AND Parallelism and Nondeterminism in Logic Programs, *New Generation Computing 3(1985)*, pp. 43-70, OHMSHA,LTD. and Springer-Verlag, 1985.
- [3] V. Kumar and C. Wang, Exploiting OR Parallelism in Logic Programs, *Under preparation*, University of Texas at Austin, 1986.
- [4] Y. Lin and V. Kumar, A Parallel Execution Scheme for Exploiting AND-parallelism of Logic Programs, to appear in, *Proceedings of 1986 Parallel Processing Conference*, August 1986.
- [5] Y. Lin and V. Kumar, A Decentralized Model for Executing Logic Programs in Parallel, *Submitted for Publication*, March 1986.
- [6] Y.J. Lin, V. Kumar, and C. Leung, An Intelligent Backtracking Algorithm for Parallel Execution of Logic Programs, *to be presented at the Third International Conference on Logic Programming*, London, England, July, 1986.

Real Time Artificial Intelligence Architecture

Peter E. Green

Ronald J Juels

Worcester Polytechnic Institute
Department of Electrical Engineering
Worcester, Mass. 01609

William R. Michalson

Raytheon Corporation
Equipment Division
Sudbury, Mass. 01776

Introduction

The research reported here is part of an overall research project into real-time intelligent systems. The goal of this research is to learn how to build the next generation of systems which have extensive embedded decision making capability. The trend is to embed ever increasing amounts of computational power into systems and to transfer responsibility for decision making from the human operator to the computers embedded within the system. The goal is to be able to build highly complex systems that are reliable and fault tolerant and yet can be operated by people who do not need a detailed knowledge of how the system works. Specifically the research is concerned with the development of hardware and software structures for real-time resource-limited decision making by computer.

Research Models

Real-time intelligent systems are being investigated by building several model systems and looking for common themes. From these themes a software architecture is being developed that can support real-time intelligent decision making and a hardware architecture is being developed to efficiently execute the required software paradigms.

The decision making systems currently under development include:

- (1) An intelligent robot which attempts to dynamically optimize the order in which it places piece parts from a conveyer belt into a set of bins. As the arm has limited motion velocities, the order in which the pieces are placed in the bins is critically important if a piece is not to pass out of the reach of the arm before it is picked up. We are using this problem to investigate the use of incremental evidence and approximate decision making in real-time computation resource limited situations.
- (2) A mobile robot which attempts to navigate its way to a goal point some distance away in a maze. The robot does not have any a priori knowledge about the maze and must learn about the maze using a rangefinding sensor which can be pointed in any direction. All decisions must take account of the time to compute the decision and of the robot's movement inertia, e.g. the decision to stop must be made in time to avoid hitting a wall.

- (3) A fault evaluator and a process scheduler for a multi-processor computer system. This program, which must run very fast and be aware of its own time-line, will evaluate fault reports from different processors and decide dynamically which processes should run on which processors. This is similar in nature to (1) above but has much more severe computational constraints and a much more dynamic environment. It also addresses the problems of distributed decision making and the difficulties caused by false reports and erroneous decisions.

Activation Frame

As a result of these investigations, a real-time decision making technique^{1,2} has been developed that performs planning by using a number of concurrent processes that execute in parallel. Some of the processes are able to quickly, but not accurately, evaluate alternative decisions and others take longer to provide a more in-depth evaluation. A decision is made at a required point in time by looking at the weight of the evidence generated by the different planning procedures up to that time. This technique has been combined with prior work in neural network modeling, real-time operating systems, and frame based AI systems to develop the Activation Framework concept³ for programming real-time artificial intelligence applications.

This technique is based on the use of Communicating Expert Objects (CEOs). Each object contains a local domain of expertise embodied in hypotheses and procedural knowledge. These objects communicate by means of messages and can execute in parallel on different processors. When a message arrives at a CEO, it triggers the execution of a section of code which may create new hypotheses, modify the evidence for existing hypotheses, and/or delete hypotheses. In addition the section of code may send messages to other expert modules to inform them of some deduction that has been made or to request information about certain hypotheses.

In application, separate CEOs could be used for each of the methods used to evaluate the evidence for different courses of action. Each of these CEOs would be sent a message containing information about the decisions to be evaluated and the world state in which they would be applied. The CEOs will then evaluate the decisions and send to another CEO the evidence for or against the proposed alternate courses of action. This CEO will then make a decision based on the evidence received from the other CEOs.

The power of this method lies in the concept that some CEOs will contain quick rules of thumb that are able to execute quickly and generate early evidence. Other CEOs may contain more exact evaluation procedures that take much longer to execute and these may involve the interaction with other CEOs that have specific domain expertise needed to support the evaluation. As the decision making CEO receives evidence it can evaluate whether it has enough evidence for a decision, or it can choose the best decision based on its knowledge of the available time before a decision must be made. In order to realize the potential of this method, an architecture is needed which is capable of supporting the execution of many CEOs in parallel. This lead to the evolution of the concepts presented here.

Any CEO can communicate with any other and does so by means of a message addressed to the destination CEO. These messages have a message type that is used to designate which section of the procedural code within the CEO will be triggered by the message. Messages also have an activation level (akin to a priority level) that is a product of the importance that the CEO places on the message and the decision making importance of the code within the CEO. To realize this software architecture, a hardware architecture was selected in which each CEO would be executed within its

own processor so that each CEO could execute truly in parallel. In making this choice it was realized that at any time many of the CEOs and therefore processors would be idle. It was postulated that such a system would still be a very cost effective decision making processor despite this inefficiency because:

- (a) This inefficiency would be more than compensated for by the efficiencies gained by not having any multi-processing scheduling overhead in each of the processing elements,
- (b) This design would have the potential for being realized at a low cost per processing element through the use of VLSI technologies.

A major thrust of our work is to investigate such an architecture.

Activation Cells

Activation Cell Processor (ACP) was chosen for the processing element that would contain each CEO. It was decided that, as there would be many ACPs in a system (our thinking number is 1024), the simplest interconnect network would be some form of ring. Each ACP, and therefore CEO, would have a unique address on the ring and all messages would pass through each ACP in the manner of a token ring. In this sense the system is similar to some data flow architectures⁴ but performs a very different function in that its major objective is decision making rather than numerical computation. As currently envisaged, the ring of ACPs will have a host interface by means of which messages can be injected into the system to cause it to start making decisions and by means of which decision results can be extracted. It is also envisaged that messages will be used to load CEO code into each ACP and to perform debugging and monitoring functions.

Summary

The implementation of this activation framework architecture for real-time AI is a major thrust of our current research. Devising a software methodology for programming CEO based expert systems and finding algorithms to efficiently handle the large volume of messages in this type of system is an important part of our research. We are also investigating an architecture for an ACP which uses specialized VLSI chips in conjunction with standard VLSI processor components to provide an efficient environment for supporting the activation framework. We are also pursuing the design and simulation of a simple bit-serial VLSI ACP for use as a test vehicle for the activation framework concept.

References

1. P. E. Green, "Issues in the Application of Artificial Intelligence Techniques to Real-Time Robotic Systems," *To appear in Proc. 1986 ASME Computers in Engineering Conference*, Chicago, IL (July 1986).
2. P. E. Green, "Resource Limitation Issues in Real-Time Intelligent Systems," *To appear in Proc. SPIE Conf. on Applications of Artificial Intelligence III*, Orlando, FL 635 (April 1986).
3. P. E. Green, "AF: A Framework for Real-Time Distributed Cooperative Problem Solving," *Collected Papers of the 1985 Distributed AI Workshop, Sea Ranch, CA*, pp. 337-356 (November 1985).
4. I. Watson and J. R. Gurd, "A Practical Data Flow Computer," *IEEE Computer* 15(2), pp. 51-57 (February 1982).

Session 15A: Miscellaneous

Chairperson: H. J. Siegel
Purdue University

A STRATEGY FOR FAILURE PREDICTION

Dorothy M. Andrews

Advanced Decision Systems
201 San Antonio Circle, Suite 286
Mountain View, CA 94040-1270

Center for Reliable Computing
Stanford University
Stanford, CA 94305-4053

There is no "problem area" more important than assurance of reliable computer systems. Unfortunately, reliability is becoming an even more crucial problem because of the criticality of many computer applications. One of the reasons for the escalating interest in reliability is that some of the most critical applications will be executed in distributed computer environments and distribution introduces another level of complexity. The preliminary results of a recent research study indicated that failure prediction might be a viable approach to increasing computer reliability. It also appears that prediction could be especially useful in increasing the reliability of distributed systems.

The main objective of a failure prediction system is to prevent catastrophic failures; however, a secondary objective is to provide early (and automated) diagnosis of possible component failure. Early diagnosis of potential problems allows recovery procedures to be initiated sooner and thereby improve the possibility of preventing failures. The sooner reconfiguration is enabled (through early warning of potential failure) for systems having distributed architectures, the less chance there is that data corruption and other related distributed system problems will occur.

The failure prediction research was part of a study to measure and model computer reliability as affected by system activity [Iyer 82,85], [Mourad 85], [Velardi 84]. Data was collected from several generations of mainframe computers. During analysis of the computer failure and activity data, Velardi observed that immediately before a crash there was an increase in the number of errors recorded by the operating system. Subsequently, part of the research effort was directed toward determining the characteristics of the change in error generation prior to failure with the hope of being able to provide a statistical basis for a failure prediction strategy. The first step was to characterize and cluster "crashes" to find appropriate intervals of time for analysis of data. The next steps involved averaging and weighting error distribution data, analyzing individual intervals between crashes, and analyzing CPU utilization rates prior to crashes. A detailed methodology for analysis of failure prediction data was developed and reported in [Nassar 85].

Since the results of a rigorous data analysis confirmed the existence of certain patterns in error distribution before a crash, a prototype of an actual failure prediction software system was implemented. The algorithm was based on the detection of large error

clusters, that is, a threshold number of errors which might be precursors of a crash. (Threshold numbers were obtained for each type of error from historical data.) When this elementary strategy was tested by simulation using available data as input, the results indicated that failure prediction may indeed be feasible. Other parameters (such as the gradual increase in error generation rate and/or system utilization) could be included in the algorithm to further refine and improve the accuracy of the strategy. Additional experimentation and study, however, is necessary to provide the ultimate assessment of failure prediction feasibility.

The failure prediction strategy proposed as a result of this research is based on actual error statistics (and probabilities) and could be implemented using artificial intelligence technologies. A particularly useful feature would be to build an adaptive or learning algorithm into the implementation of a failure prediction strategy in order to improve diagnostic capabilities. If this were done, then changes in architectures of the host computer would not require a completely new implementation of a failure prediction strategy.

This work was supported in part by the U.S. Army Research Office under contract number DAAG29-82-K-105. The views, opinions, and/or findings contained in this document are those of the author and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other official documentation.

[Iyer 85] R.K. Iyer and P. Velardi, "Hardware Related Software Errors: Measurement and Analysis," *IEEE Transactions on Software Engineering*, Feb. 1985.

[Iyer 82] R.K. Iyer, S.E. Butner, and E.J. McCluskey, "A Statistical Failure Load Relationship: Results of a Multicomputer Study," *IEEE Transactions on Computers*, July 1982.

[Mourad 85] S. Mourad and D.M. Andrews, "On the Reliability of the IBM MVS/XA," *Proc. The Fifteenth Int'l Symposium on Fault-Tolerant Computing (FTCS-15)*, Ann Arbor, Michigan, June 19-21, 1985.

[Nassar 85] F. Nassar and D.M. Andrews, "A Methodology for Analysis of Failure Prediction Data," *Proc. Real-Time Systems Symposium*, San Diego, CA, Dec. 1985.

Velardi 84 P. Velardi and R.K. Iyer, "Software Related Failures and Recovery in the MVS Operating System," *IEEE Transactions on Computers*, Vol c-33, No. 6, June 1984.

Programming Language Translation for Multicomputers

Jon Mauney

Computer Science Department
North Carolina State University

Everyone seems to agree that parallel computing is an essential part of the future, and that programming is a major challenge in effective use of parallel systems. The problem for language and compiler researchers in the coming years is to provide languages convenient to programming and compilers to translate them to efficient parallel code.

We can divide the problem of compiling for parallel target machines into two phases: parallelism detection and parallelism exploitation. Detection is the process of discovering which activities may be performed in parallel; exploitation is the process of determining which activities will in fact be performed in parallel on the limited resources of the machine. Parallelism detection can be performed without regard to the characteristics of the target machine; parallelism exploitation is dependent on the machine, but is independent of the language in which the program was written.

Although detection and exploitation can be separated, they are typically considered together, and most papers present a technique for a particular language and target machine. Vectorizing compilers, for example, only search for the kinds of parallelism that a vector machine can exploit. The trace scheduling technique is closely associated with the Bulldog compiler and the VLIW machine[2].

Although there is nothing wrong with combining the two phases in a particular compiler, there is something to be gained by separating them. Just as traditional optimization techniques for sequential machines are divided into machine-independent and machine-dependent parts, so should optimizations for parallel machines be divided into machine-independent parallelism detection, and machine-dependent parallelism exploitation. Future work on parallel compilers will benefit by maintaining the distinction.

When one discusses parallel compilers, one of the first questions asked, unfortunately, is "what language will you compile?" The choice of language is, of course, very important. Features of the language may help or hinder the compiler in the discovery of parallelism. More importantly, a good language for parallel computation will encourage the programmer to write code so that parallelism inherent in the algorithm is not obscured, and perhaps to think of algorithms that have more inherent parallelism. But the choice of language really only affects parallelism detection, and the emphasis on languages tends to draw attention away from the equally important problem of parallelism exploitation.

A parallel loop may be written as a sequential for-loop, with the parallelism being detected by careful analysis of data flow, or it may be written as an explicitly

parallel forall loop, or it may be written as a single operation to an entire array, as would be typical in APL. Having discovered that the loop is potentially parallel, the compiler must still determine whether the parallelism can be used by the target machine, and whether using the parallelism is likely improve the performance of the program. If parallel computation increases contention for resources or overhead for synchronization, then the speedup due to parallelism may be lost. On some machines, careful selection of parallel operations and assignment to processors and memory is as important as detecting the parallelism in the first place.

Since parallel compilation techniques are usually presented in the context of a particular model of parallel computation, their potential applicability to other models of computation is often given little notice. Loop analysis techniques are most often applied to vector and array processors, but the parallelism detection they provide may be used on a variety of machines. Trace scheduling is associated with the VLIW machine [2], but may be useful on other multiprocessor systems [1]. By paying attention to what the machine dependencies really are in various compilation techniques, researchers will be better able to share and adapt good strategies. The sharing of similar strategies for parallelism detection and exploitation will also facilitate comparison of various models of parallel execution.

Carefully maintaining the distinction between the machine independent and dependent parts of a parallel compiler, then, will aid future work in several ways. First, it will help to emphasize that the goal of a new language for parallel computing is not to make the programmer code in more parallelism, but to keep him from coding in sequentiality. Ideally, the programmer should be removed from low level details of parallelism. Second, it will help to show the similarities and differences in various compiling techniques. And third, it will help to make software support more uniform across a variety of parallel architectures, simplifying the problem of comparing their merits.

1. References

1. Z. Amir, Decomposition of a source program into parallel components, Master's Thesis, N.C. State University, 1985.
2. J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures*, MIT Press, Cambridge, 1986.

Resilient Procedures: A Structured Replication Approach

Kwei-Jay Lin
Mark E. Wittle

Department of Computer Science
University of Illinois at Urbana-Champaign
1304 West Springfield Avenue
Urbana, Illinois 61801

1. Availability and Replication

Distributed systems have the potential to achieve better reliability and performance because of their many resources. Atomic actions [7] have been proposed to guarantee correct system function regardless of site crashes and conflicts between concurrent operations. By ensuring the atomicity of a Remote Procedure Call (RPC) [6], atomic RPC [5] can provide a transparent and reliable primitive which hides distributed programming complications from naive users.

One desired property which is not provided by atomic RPC is *availability*: the system should be available with high probability. *Replication* [4] has been proposed as a means to increase availability; either data or operations are replicated on independent sites so that some sites' failures still leave the system with enough resources to continue execution. Replicated data are stored on different sites and are usually identified by version numbers. Algorithms exist to guarantee consistency between copies [2,3]. Due to the recent dramatic decrease in the cost of computer hardware, it is now possible and will be common in the future to have many processors devoted to a single application. We are implementing a new construct, Resilient Procedure (RP), which allows a procedure to be hierarchically replicated at several sites so that the service provided by the procedure will be available despite site failures. The RP construct has several desirable properties:

- (1) **Fault-Tolerance:** By replicating a procedure at k sites, the resulting RP can survive up to $k-1$ site failures.
- (2) **Transparency:** From a caller's point of view, a RP is the same as a normal procedure providing the same service. Thus

a normal procedure can be replaced by a RP whenever the resilience or the concurrency is desired.

- (3) **Efficiency:** RP's may yield performance gains, as some sites may be less loaded, providing shorter response time than if the procedure were performed at a single site.

In Section 2, we briefly describe the RP construct. We discuss some fault tolerance issues in Section 3.

2. Resilient Procedure

We assume first that all sites are homogeneous. Heterogeneous environment problems will be discussed in section 3. Networks are assumed to be asynchronous but reliable, i.e. messages are always delivered correctly and in order.

A RP is a special procedure construct which consists of a *coordinator* and several *cohorts* (Figure 1) residing on multiple sites. The coordinator is the only entity of the RP visible from outside. All incoming call/return messages are directed to the coordinator and all outgoing call/return messages are sent from the coordinator.

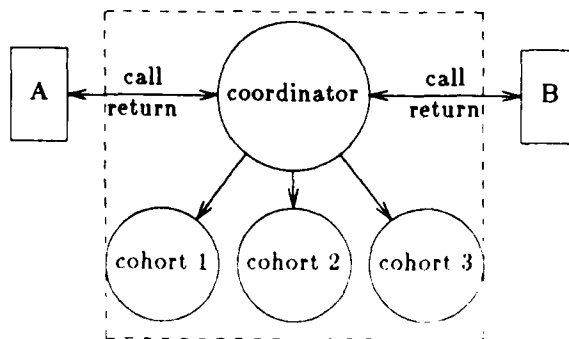


Figure 1 RP called by A and calls B

This research was supported by the University of Illinois Research Board under Grant 1-2-69730-6300.

When an incoming call message is received by the coordinator, it determines if the call can be accepted, just like a normal atomic procedure. If the call is accepted, it is then transferred to the cohorts. Upon receiving the call message from the coordinator, each cohort starts its execution independently. No global coordination is required.

When a cohort wants to call an external procedure, instead of sending the call message to the callee directly, the call is sent via the coordinator. The coordinator collects all cohort call messages destined for the callee and sends only one call. The fact that there are multiple cohorts is transparent to the external procedure which performs a single call as if it were from a non-resilient procedure.

The coordinator does not have to wait until all of its cohorts have requested the call to the same callee. Although some cohorts may be slower than others, all will eventually make the call (unless they crash - this is handled separately). If a fast cohort has made the call, and the result has already been returned, then the coordinator simply returns that result to the slower cohort immediately. This approach, besides speeding up the overall performance, synchronizes all cohorts by allowing a slow cohort immediate access to external results.

When an external procedure call result is received by the coordinator, it checks to see which cohorts are waiting for the result. The result is distributed to those cohorts which then resume execution. Upon completion, each cohort returns its result to the coordinator. The coordinator collects results from all cohorts and returns a single result to the caller.

3. Fault-Tolerance Issues

3.1. Atomicity Considerations

Concurrency control is required to implement atomic procedure calls. One procedure may receive call messages from several concurrent caller procedures; only one of the calls should be accepted to avoid inconsistent read/write interleaving on local data. Furthermore, if a nested atomic call model is used, a non-recursive new call must be delayed until the previous atomic call has committed. Since the coordinator is the only entity visible from other procedures, it must perform concurrency control for the RP.

Two types of failures can occur during a procedure's execution: *expected* or *unexpected* failures. An expected failure can be handled by some kind of exception handling mechanisms. An unexpected failure is usually handled by a backward recovery mechanism. In the RP environment they can be treated as site crashes, merely reducing the number of cohorts in the RP. However, if we can guarantee that each RP always has at least one active cohort available and if all procedures in the system are implemented with the RP construct, then no backward recovery is needed.

3.2. The Coordinator Resiliency

The coordinator is the critical component of the RP; its failure renders the cohorts unreachable. Therefore, if the coordinator crashes, one of the cohorts must be promoted to become the new coordinator for the RP. In order to allow a cohort to become a coordinator, any system information maintained by the coordinator must be replicated in each of the cohorts. Our approach is to piggyback updates to the system information table onto the call request and return messages exchanged between the coordinator and its cohorts. Since the system information is updated only when there are new call/return activities, there is always communication between the coordinator and cohorts immediately following a table update. So the coordinator does not have to wait a long time before the update is propagated.

When the coordinator crashes, the cohort which communicated with the coordinator most recently has the current system table, and will become the new coordinator. Each cohort keeps a timestamp sent by the coordinator during the last successful conversation. By comparing timestamps in a voting protocol, the cohort with the latest timestamp is promoted.

3.3. Heterogeneous Environment

Cohorts running in a heterogeneous environment can return different results with different response times. Depending on the application, many protocols can be used to choose one result from the returned values. Some of the possible protocols are: *majority*, *first*, or *primary*. *Majority* protocol takes the result of the majority of the cohorts and is used to deliver more reliable results in a noisy environment. *First* protocol returns the first result received by the coordinator

and is used to improve performance. Primary result protocol is used if one cohort is known to be the most accurate or reliable.

A heterogeneous environment can be used to detect faults at cohort sites. By replicating a procedure onto multiple sites, the RP becomes tolerant of transient physical faults. If the sites are heterogeneous, the RP then provides hardware design tolerance. Software design tolerance for the system software and execution environments at the distributed sites is also provided. Furthermore, if different software versions [1] are executed at different cohorts, software fault-tolerance can also be achieved.

The RP construct allows the user a great degree of freedom in choosing the method used by the coordinator to resolve inconsistent cohort execution results. The advantages of a consensus decision are imparted by replication, and application specific criteria can also be incorporated into the decision algorithm.

4. Conclusion

In this paper, we presented a new system construct, Resilient Procedures, which has the desirable properties of fault-tolerance and availability. We discussed the atomicity and heterogeneous issues that can occur when replicating a procedure. We showed further that the structure can be used for many applications in distributed and parallel computations.

By allowing the user to specify the degree of replication, the diversity of replications (e.g. heterogeneous sites, software versions, etc.), and the method of coordination, the RP construct provides the user a friendly and powerful programming environment.

References

- (1) Avizienis, A., "The n-version approach to fault-tolerant software," in *IEEE Trans. Software Eng.*, Vol. SE-11, no. 12, pp. 1491-1501, Dec. 1985.
- (2) Bernstein, P.A., and Goodman, N., "An algorithm for concurrency control and recovery in replicated distributed databases," *ACM Trans. Database Systems*, Vol. 9, no. 4, pp. 596-615, Dec. 1984.
- (3) Gifford, D.K., "Weighted voting for replicated data," in *Proc. 7th Symp. Operating System Principles*, December 1979.
- (4) Herlihy, M., "Atomicity vs. availability: Concurrency control for replicated data," Carnegie-Mellon Univ., Pittsburgh, PA, CMU-CS-85-108.
- (5) Lin, K.-J. and Gannon, J.D., "Atomic remote procedure call," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 1126-1135, Oct. 1985.
- (6) Nelson, B., "Remote procedure call," Ph.D. Dissertation, Carnegie-Mellon Univ., Pittsburgh, PA, CMU-CS-81-119.
- (7) Reed, D., "Implementing atomic actions on decentralized data," *ACM Trans. Comput. Syst.*, vol. 1, pp. 3-23, Feb. 1983.

Session 15B: VLSI Related Issues

Chairperson: Mary Jane Irwin
Institute for Defence Analysis

PARALLELISM AT THE MICROLEVEL: COOPERATIVE MICROCONTROLLERS WITH REAL TIME CONSIDERATIONS

Christos A. Papachristou
Computer Engineering and Science
Center for Automation and Intelligence Systems
Case Western Reserve University
Cleveland, Ohio 44106

Introduction

Increased computer power and performance in recent years made possible advanced applications in such fields as avionics systems, aerodynamic simulations, industrial automation, military defense, weather forecasting, and more. Most of these advances are based on the concept of parallel processing. Parallelism can be applied at various levels of computation, i.e., the algorithm level, program level, system level, machine level, and the microlevel. Research effort on parallel processing currently underway has been directed mostly at the higher end of the computation spectrum. For example, work on multiprocessing, pipeline machines and distributed computers is well documented in the literature. Little has been done at the lower end, with the exception perhaps of systolic array structures.

The aim of this work is to propose an approach to low level parallel processing. In our view, parallelism at the microlevel refers to a multiprocessing environment controlled by cooperative microengines executing concurrent microcode. This environment provides the capability for parallel implementation of functions via concurrent microprogramming. This would essentially constitute parallel firmware migration. The reasons for parallel migration are quite similar to the ones of function migration in uniprocessing structures; namely, substantial gains in speed, reliability and stability. Moreover, due to faster time response, microlevel parallelism would be better equipped to handle real-time applications, e.g., multisensor functioning of complex automation systems. In what follows, first we present the essential ideas of a new technique for function migration in a uniprocessor controlled environment. Then, we describe the structure of a cooperative microcontrol system together with extended sequencing protocols to support concurrency at the microlevel.

Migration Technique

Recall that vertical migration of frequently used software into firmware, labeled firmware migration, is a well known technique for improving the system performance. It has been practiced by the designers of such systems as the IBM 370 and the DEC PDP 11 series. However, firmware migration has been influenced by VLSI technology due to the capability to embed in silicon not just "traditional" microprograms, i.e. instruction set interpretations, but also complicated software functions such as data tables, parsers or operating system primitives. In addition to system software, complex algorithms appear in other areas that may have military applications such as communication algorithms and radar tracking. In general, such functions have complex logical, i.e. sequencing, structure. They are represented by complex flow charts with many multiway branchings, nested loopings and the like. Thus, cost-effective migration of such functions requires modular microprogram structures with powerful sequencing capability.

Our major objective of our recent work has been to explore a new methodology for automated software-to-firmware migration based on a new microcontrol architecture we have developed recently. This scheme is endowed with complex sequencing constructs available at the microlevel. We feel that our approach has the advantage of compatibility with VLSI in that it effects software migration into VLSI microcode via firmware techniques. Such an automated migrator could be an important ingredient of a hierarchical silicon compiler if it is interfaced to appropriate CAD layout tools.

The basic idea of the proposed method is to capture the sequencing structure of the candidate function via effective compilation techniques. This idea can be described by the following abstract migration model which, incidentally, can be employed at several computation levels but it is very effective at the microlevel. Every function F can be represented by the following two data structures:

1. A sequencing graph, dependent on F .
2. A library of operation modules, which is independent of F .

The sequencing concept encapsulates essentially the sequencing structure of F . It consists of a listing of pure sequencing constructs including sequence calls to the library. The modules in the library consist of series of command-types performing related operations comprising a meaningful subfunction, for example ADD module. Specifically, the modules comprise the structured firmware implementation of an instruction set of a base machine on which the migrating function is to be tested. If this firmware code is not already available, it may be produced by a microcode generator tool and executed on the base machine.

The important point here is that the library is a universal structure, i.e., it should be general enough to cover a collection of functions but, once it is written, it can be employed with every function in the collection. To perform migration, in this model, we only need to generate the sequencing structure of each F and then link it to the library. The basic steps to do this are: a) function high-level compilation, b) control flow graph construction, and c) sequencing constructs detection in the function graph. The end result is the generation of microsequencing code for the function, consisting of the constructs mentioned earlier, particularly sequence calls to the library of microcode modules.

An important requirement for the proposed method to be practical is the retargetability of the migration process. This term means the adaptability of firmware migration to a new base machine obtained by architecture redefinition. The latter is usually accomplished by means of a hardware design language at the register-transfer level. In our view the following three problems are involved:

- (a) Redefinition of the base machine structure.
- (b) Retargetability of the migrator sequencing code.
- (c) Library construction for the machine defined.

Our approach involves the use of a hardware design language (HDL) such as ISP-N.mPc developed by other colleagues at Case. The basic property of these languages is their capability to define a microarchitecture by its structure and behavior. We plan to expand this capability by defining the control scheme to be used. Our basic idea to achieve retargetability is to capture the assembly code version of the migrating function and then proceed along the steps described earlier. However, we propose to use as assembly code the behavioral statements of the HDL. Further, we plan to use a preprocessor which will translate the high-level language description of the migrating function into the HDL behavioral code. Machine dependencies will be introduced at that level by binding together

the structural and behavioral descriptions of the machine. A major advantage of this technique is that it does not require changes in the migration process when the base machine is redefined. Thus, the problem of migration retargetability is handled by the means used for architectureecification.

Parallelism at the Microlevel

In general, parallel processing at the higher-end refers to the concurrent operation of several processors which share a number of resources. Clearly, there should be a control mechanism or protocol to regulate the accessing and releasing of resources to avoid conflicts. Such regulation mechanisms based on mutual exclusion principles are well documented in the literature. These general principles also apply to parallelism at the microlevel. In our approach, we consider a cooperative microcontrol environment consisting of several microcontrollers each executing a migrating function. The proposed scheme is briefly discussed below.

Consider m microcontrollers C_1, \dots, C_m , that implement concurrently the migration functions F_1, \dots, F_m , ictively. More precisely, each C_i contains the sequencing structure of F_i in its sequencing store S_i . Moreover, there exists a library L_i of microcode modules embedded in the microcode store M_i of C_i , $i=1, \dots, m$. There are also n processing regions P_1, \dots, P_n which can be requested by the microcontrollers such that P_i can be requested by only one microcontroller C_j at the time. We make no assumption as to the nature of processing regions P_i , in other words, they may be processors, peripherals, memories or simply distributed hardware. The requests of the processing regions are made by explicit reference of the regions in the code structure of the library microcode modules. For example, an ADD type of module in the second library L_2 may refer to an adder located in the third processing region P_3 . These references of microcode modules to regions may be static or dynamic. The preceding adder example is a static reference whereas an indirect reference to a main memory location is dynamic.

The main problem of the cooperative microcontrol scheme is a resource management problem at the microlevel: the effective regulation of the competitive requests of processing regions coming from microcode modules. Although solutions to this problem at the high end are well documented in the literature, the problem has not even been formulated in the microlevel context. The advantage of our approach is in utilizing a microcontrol scheme with powerful sequencing capability supported by developed tools. We feel that for cooperative microcontrol we need to extend the proposed sequencing constructs to include constructs that provide synchronization at the microlevel. The coordination protocol is conceptually described below, assuming that each microcode module requests a single processing region at each time.

When the microcode module Dip , located in library L_i , is invoked by a sequence call, then the controller C_i should issue a signal indicating the attempt to "enter" (request) a processing region P_k . Again, P_k is referenced in the code structure of Dip . If P_k is not currently "occupied" i.e. not requested by another module D_{jq} , then Dip can safely enter P_k . However, if P_k is currently occupied, then C_i should enter a "wait-until" loop till P_k is "released." During this waiting time the microcode store should be in a NO-OP state, perhaps via a No-Op module. Additional sequencing constructs required are "enter region" "exit region", "return without exit" and "return with exit". These should be included in the repertoire of constructs for microlevel implementation of parallelism in the context of our scheme.

WAFER SCALE IMPLEMENTATION OF A GaAs SYSTOLIC
SIGNAL PROCESSOR CELL

H. Merchant, H. Greub, R. Schreiber,
Capt. B. Donlan and J. F. McDonald
Center for Integrated Electronics
Rensselaer Polytechnic Institute
Troy, New York 12181
(518) 270-6033

ABSTRACT

This paper describes some preliminary results obtained by the authors while attempting to design a systolic signal processing element to be implemented using the Tektronix-TRIQUINT GaAs E/D MESFET Foundry.

INTRODUCTION

The design constraints implied by the literature published by TRIQUINT for their E/D MESFET gates are:

- 1) Fanin ≤ 3
- 2) Fanout ≤ 4
- 3) Number of gates per testable cell ≤ 80 (for 83% yield)

The goal is to design a systolic cell similar to that implemented by the Saxby Computer Corporation but using GaAs. Because of the yield problems of GaAs, the architecture had to be decomposed into small "bit-slice" cells which could be fabricated, tested and interconnected to form the systolic element. The interconnection process employs fine geometry (5-10 μm) thick metal lines embedded in a polyimide dielectric. The logic cells can be "in" the substrate or "on" the substrate.

Given the space constraints of this article, here we consider only the implementation of the systolic cell function $Y=X+A*B$ in 16-bit arithmetic. Since the TRIQUINT process involves 1 μm feature sizes, their lithography employs step and repeat reticles (up to 4 may be specified). Figure 1 shows the bit slice cells defined as module subdivisions of these reticles. The four modules $M_1 - M_4$ are various column counters for the multiplier and the S cells are all identical four-bit carry select adders.

If these cells were fabricated and mounted in separate packages, the packages would adversely affect the performance of the system. By directly connecting the dies on the wafer (wafer scale integration or wafer scale hybrid) an extremely dense system is feasible.

Figure 2 shows the overall architecture of this systolic cell. Figure 3 shows a simulated four-inch wafer with several

hundred (at 70% yield) of the desired cells in the rough proportions demanded by the architecture. Figure 4 shows the routed wafer. Evidently, there will be plenty of room to scale up the processor to a full 32-bit processor possibly with floating point features.

A histogram of the typical wiring lengths required for the 16-bit integer case is shown in Figure 5. The estimated maximum delay for this version of the systolic cell is 7.92 ns plus I/O pad delays. Improved versions are definitely possible, especially if the TRIQUINT yields are even only slightly better.

CONCLUSION

Wafer integration of many small GaAs E/D MESFET cells offers an attractive alternative to VLSI when the yield is low for the latter.

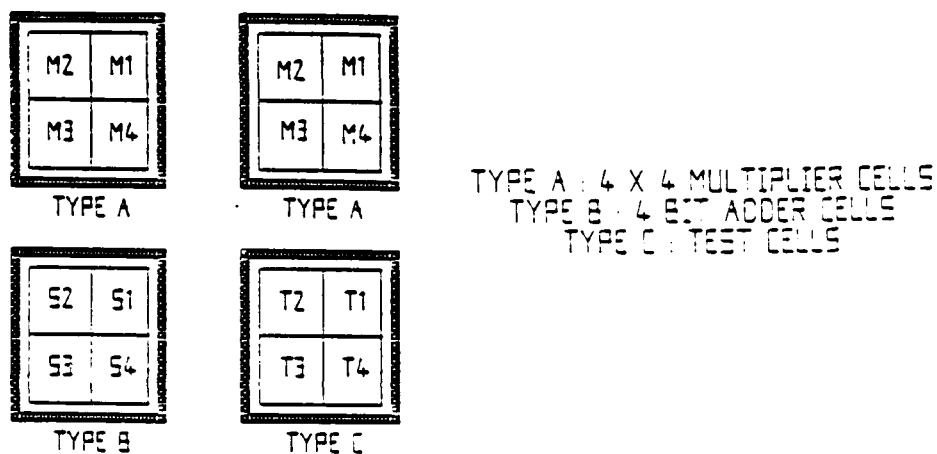


Figure 1. Step-and-Repeat Reticle Module Assignments

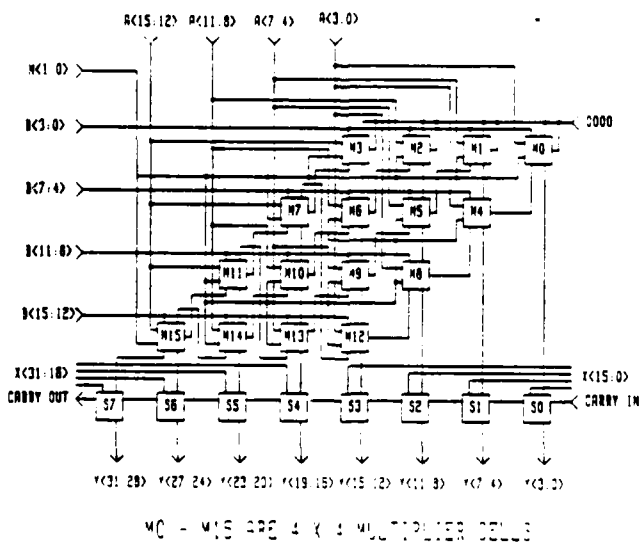


Figure 2. Overall Systolic Cell Architecture

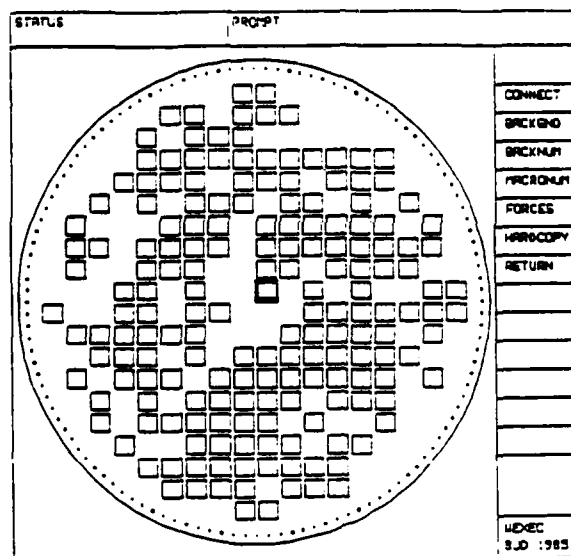


Figure 3. Four-inch diameter wafer with dies at 70% yield

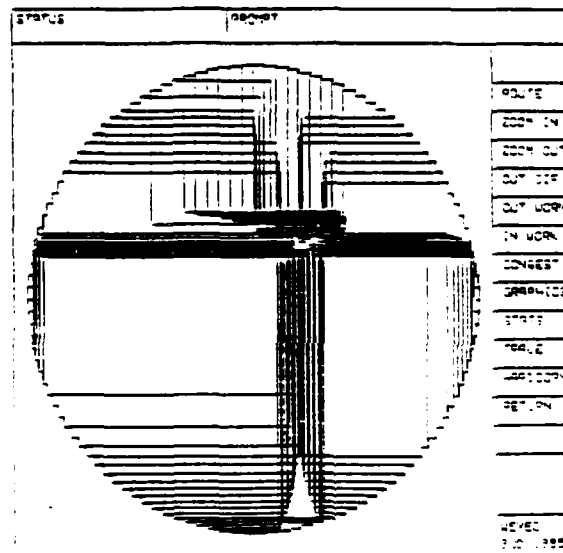


Figure 4. Routed wafer for systolic architecture

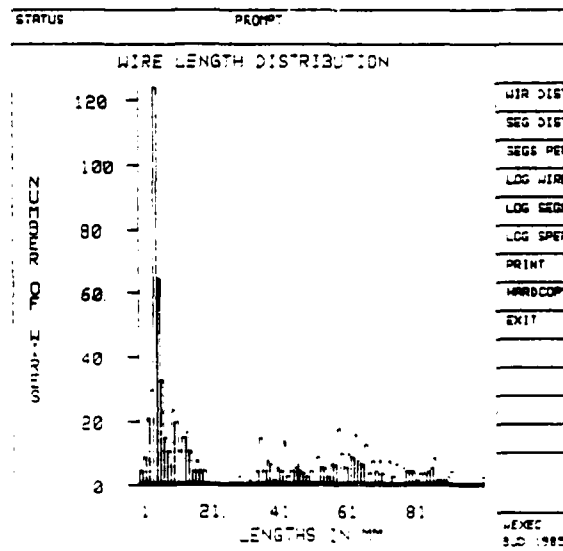


Figure 5. Histogram of wire lengths used in architecture

DESIGN FOR A TPL COMPILER SYSTEM -- A SYSTEM FOR RETARGETING HIGH LEVEL LANGUAGE PROGRAMS

S. Leong, O. Jiang, S. Jodis, and P.A.D. de Maine

Computer Science and Engineering Department
Auburn University
Auburn, Alabama 36849

I. INTRODUCTION:

A key problem is the retargeting of software coded in high level languages so that they will efficiently execute in new computer environments that may have totally different computer architectures and/or supporting software. A standard high level language like ADA [1] that is fully transportable cannot, by itself, solve the fundamental problem of transporting existing programs to new environments. Mimicking, either by hardware or software, the source environment is not a viable solution because such methods [2]: Must be separately implemented for every different object environment; and they do not fully exploit the characteristics of the new environments.

In the Transportable Programming Language, TPL, Method [3] the essential concept is a retargetable bifunctional compiler, called the HLL-TPL Compiler, that uses a hypothetical language, called the Hypothetical Parent High Level Language (HPHLL), as the transporting or retargeting vehicle. The HLL-TPL Compiler, which can be retargeted for any environment (or dialect), converts the local dialect to the HPHLL and visa-versa. That means the TPL System, which consists of the HLL-TPL Compiler and its supporting software, can be transported or retargeted by itself.

The TPL system that has evolved has three different parts:

1. The Support Facilities
2. The Hypothetical Parent High Level Language, HPHLL
3. The HLL-TPL Compiler System

The Support Facilities are used by the bifunctional HLL-TPL Compiler to compensate for:

- (a) Differences in architectural features like the byte or word sizes and the amount of available core-memory.
- (b) Special instructions that exploit architectural features, like the character type operation in IBM FORTRAN.

The Hypothetical Parent High Level Language, HPHLL, for every different high level language is constructed from its dialects as described in [3]. Essentially the HPHLL is a superset that contains all the instructions in all the dialects of the HLL.

The HLL-TPL Compiler System that is now being tested has three parts :

1. The Rule Modifier.

2. The Table Generator.
3. The HLL-TPL Compiler itself.

An essential part of the design is the Conversion Rules Description Language, CRDL [4]. CRDL is a meta-language that is used to describe the procedures for converting one language into another. The CRDL descriptions of the conversion procedures are used by the Table Generator to produce the TTM and CM Tables that drive the HLL-TPL Compiler in its two modes: For converting the local dialect to HPHLL and the HPHLL to the local dialect.

The Rule Modifier and Table Generator are normally used only when the HLL-TPL System is installed, while the HLL-TPL Compiler is used for each program conversion. The roles of the Rule Modifier, Table Generator and HLL-TPL compiler are discussed in Sections II, III and IV.

II. RULE MODIFIER:

The Rule Modifier is a special purpose editor that accepts descriptions of differences between the Default Dialect (or environment) and the Local Dialect (or environment) and then modifies the files, coded in CRDL [4], that are used by the Table Generator to produce the tables that actually drive the HLL-TPL Compiler. The functions of the Rule Modifier are as follows.

1. It is a tool for altering the CRDL descriptions of the Default Dialect.
2. It is an abstract layer between the person who installs the HLL-TPL System and the detailed descriptions of the procedures for converting the Local (or object) Dialect to the HPHLL and vice-versa.

The description of the Default Dialect is an integral part of the HLL-TPL Compiler System. The Default Dialect is a composite constructed from the "most probable" dialects. Those charged with new installations of the HLL-TPL System use the Rule Modifier only once to alter the descriptions of the Default Dialect to the Local Dialect. It should be noted that this approach simplifies the task of installing HLL-TPL and that it is especially useful for comparatively well standardized languages like FORTRAN or COBOL.

III. TABLE GENERATOR:

The Table Generator part of the HLL-TPL Compiler System can be viewed as a special compiler that translates CRDL descriptions of the conversion procedures to a form that is used by the HLL-TPL Compiler to convert the local dialect to HPHLL and vice-versa. The Conversion Rule Description Language, CRDL, is a meta-language that acts as a second abstract layer (between language experts and the actual implementation of the HLL-TPL Compiler).

The Table Generator translates the descriptions in the intermediate files to the two so called Internal Tables (TTM and CM) that are used by the HLL-TPL Compiler to convert the local dialect to HPHLL and visa-versa.

IV. HLL-TPL COMPILER:

The HLL-TPL Compiler can be viewed as a simulated compiling machine with a dispatcher and processors to execute its own instructions. In the Transportability Testing (TTM) and Compilation (CM) Modes the HLL-TPL Compiler executes a conversion procedure by fetching instructions one at a time from the TTM and CM table respectively. The sequence of operations is terminated when the conversion procedure is completed. This kind of approach for implementing the HLL-TPL Compiler offers the following advantages:

1. Transportability is more easily achieved because the TTM and CM tables have uniform structures and both its size and complexity are significantly reduced by describing the conversion procedures in the form of easily accessed tables.
2. An efficient implementation can be realized by using some of the many well known optimization techniques [5].
3. Debugging and maintenance are simplified because the difficulties associated with debugging parsing tables are avoided [5].

The planned extension of the TPL method for use to convert any dialect of any high level language to a standard language like ADA are discussed in [4].

V. REFERENCES:

- [1] For example:
 - a. H.F. Legard, "ADA: An Introduction and the ADA Reference Manual", Springer-Verlag, New York, 356 pages (1981).
 - b. Collection of ADA papers in: Proc. of the ACM SIGPLAN Symposium on the ADA Programming Language, ACM SIGPLAN NOTICES vol 15, no 11 (1980).
- [2] P.A.D. de Maine and C.G. Davis, "Specification for a Transportable Programming Language System", Proc. COMPSAC 1982, 468-494 (1982).
- [3] P.A.D. de Maine, S. Leong and C.G. Davis, "A Transportable Programming Language (TPL) System. I. Overview", Int. J. Comp. and Inf. Sciences 14, 161-182 (1985).
- [4] S. Leong, O. Jiang, S. Jodis and P.A.D. de Maine, "A Transportable Programming Language (TPL) System. II. The Bifunctional Compiler System, Submitted to Int. J. Comp. and Inf. Sciences 3/May/1986.
- [5] A.V. Aho and J.D. Ullman, "Principles of Compiler Design", Addison-Wesley Publishing Company, Reading, Mass. U.S.A (1973).

A VLSI Implementable Block Oriented Data Driven Multiprocessor

B. Shirazi
Southern Methodist University

A.R. Hurson
Pennsylvania State University

ABSTRACT

The traditional parallel processing approaches to closing the computational gap have consistently failed since parallelism has been forced through the extensions of the sequential von-Neumann model rather than using an inherently parallel model. The data driven model allows a high degree of parallelism and asynchrony among its basic operators and thus, it has been considered as a promising alternative to the von-Neumann based architectures. This paper first discusses the trend in the design of the data driven machines through a classification of these systems. It then addresses the design of a processing module capable of executing a program block in data driven fashion. These processing modules are used as the building blocks for the construction of a block driven multiprocessor. The organization of this system matches the semantics of the block and data driven models by allowing efficient and faster intra block as opposed to the longer inter block communications. In addition to the basic design, we will address the VLSI complexity of the system by estimating the geometry area of the basic elements and the timing delay of the operations based on the current technology.

1. Introduction and Background

The challenge of closing the *computational gap* has promoted the introduction of some alternative architectures to the von Neumann machines. The *data driven model* is one the most promising of such alternatives. In a data driven environment, an operation is carried out, in parallel with other operations, as soon as its input operands are available. The concept of *asynchrony* embedded in the definition of a data driven architecture provides grounds for a high degree of implicit parallelism. In addition, the data driven organization eliminates the need for an updatable storage, use of identifiers and all of their associated by-products such as global side-effects and aliasing.

Nevertheless, the parallel nature of the data driven computation coupled with the freedom from side-effects, imply a strain on the interconnection network of these systems. This leads to the increased cost and complexity of the network as well as erecting a potential bottleneck due to network delays. Therefore, the large-grained data flow architectures have recently been investigated as an alternative for their fine-grained counterparts. This has led to the development of the block driven architectures which explore the parallelism at the program block level [1].

Basically, data driven architectures are decentralized control systems and hence, they could be classified as MIMD machines. We propose a classification which is based on the interrelationship and communication among the data memory, the instruction memory, and the processing elements. According to this classification, the data driven architectures are grouped into three classes. Class I, which is representative of the dynamic data driven machines, uses a separate data and instruction memory as well as a set of separate processing units. The Manchester and the Irvine machines are examples of the architectures in this class. They allow recursion and parallel execution of different activations of a block. However, some additional overhead is introduced due to the manipulation of the color information and the fact that only one copy of the program block is kept. The Class II, or static data driven architectures, represents the unification of the data and the instruction memories in which each instruction is self contained. The MIT and the TI machines are some examples in this group. Even though the machines in this class have a simpler organization and do not introduce additional overhead, they do not allow coexistence of multiple activations of the same block. Finally, in Class III, as a promising representative of the future data driven machines, the processing power is incorporated at the memory level. In other words, each memory cell contains an instruction and has enough processing power to carry out the operation, when the data is available. The system has a cellular organization and

therefore it is suitable for VLSI implementation.

This paper discusses the architectural aspects of a data driven organization which is based on the characteristics of the Class III, while incorporating multiprocessing capabilities at the program block level (block driven). The VLSI complexity of the proposed architecture is discussed through an analysis of the timing and geometry areas of the building block elements.

2. The System Overview

A viable data driven architecture should comply with the technological constraints, offer a better performance for inherently parallel problems, reduce fine-grained communication delays, and introduce a practical and effective solution for the manipulation of the data structures. These criteria have led us to the introduction of the *WDDM, a Wafer-scale Data Driven Multiprocessor* [2]. The system consists of m identical processing modules (each on a silicon wafer), a host, and a data structure module. The system units communicate through a double star interconnection network, with the host and the data structure modules as the centers and the processing modules as the orbital nodes (Figure 1). The two centers are connected via a dedicated channel which allows them to share the run time system management information. It should be mentioned that the choice of the star network is not fixed and we are currently investigating other networks as possible candidates.

Although the star network is potentially prone to creating a bottleneck at the center station, we have incorporated many measures in the architecture in order to alleviate the possible bottleneck. For example, a number of independent dedicated buffers are used for simultaneous transfer of blocks between the host and several processing modules. In addition, in order to reduce the overhead of the host in the memory management operations, associative processing is utilized. This not only allows fast parallel search of the different tables which are used for memory management operations, it also provides opportunities for parallelism among different associative memories.

It should be mentioned that the system operations are based on the dynamic data flow principles by allowing simultaneous execution of different activations of the same block. However, within a block, the operations are based on the static data flow model. In other words, coloring (labeling) is used at the program block level instead of the data token level.

The *data structure module* holds the data structures and partially performs or initiates the data structure operations, using the processing modules. An input data structure is duplicated for each function, thus eliminating the side-effects among the functions. However, upon the completion of a function, the input and intermediate data structures are destroyed and only a pointer to the output data structure is returned. Within a function, a scheme similar to the *I-structure* proposed by Arvind [3] is utilized. Provisions are made to utilize processing modules to allow vector operations on arrays. In addition, the interleaved organization of the data structure memory allows simultaneous access to many elements.

3. The Proposed Processing Module

An active program block assigned to a processing module is executed in data flow fashion. Therefore, there are two levels of parallelism during the execution. First, there is a large-grained parallelism due to the concurrent execution of the program blocks. Second, there is a fine-grained parallelism due to the data flow operations within a processing module. In order to reduce the network delay of the fine-grained data flow parallelism, the processing power is distributed among the memory or instruction cells. Therefore, as much of the processing as possible is carried out in the instruction cell itself and only the result values are routed to the network. However, because of the economic considerations and according to the RISC (Reduced Instruction Set Computers) philosophy, only the simple but most often executed instructions are executed by the instruction cells and the complex operations are sent to the coprocessors.

Figure 2 depicts the general organization of a processing module. The instructions of a data flow program block are distributed among the *Elementary processing units (E-units)*. The simple

operations such as fixed point addition are directly executed by the E-unit, resulting in a data token which is sent to the succeeding instruction via the sub-net. The E-units are also responsible for matching the input data tokens. For more complex operations, such as floating point division, the E-unit forms an operation token and routes it to the corresponding *Functional unit (F-unit)* through the sub-net. The data tokens generated by the F-units are routed to the destination instructions.

The *sub-net* is an arbitration network providing the intra-module communication among the elements of a processing module. It also connects a processing module to the host and the data structure modules.

The *Active/Inactive Detector (AID)* unit keeps track of the status of a program block as it is executed by the processing module. When a processing module becomes idle due to a procedure call, the status is reported to the host and the block's image is saved in the host. This allows the processing module to be assigned to another enabled block.

4. VLSI Complexity

The proposed system is suitable for VLSI implementation because it has a regular and cellular organization; i.e. duplication of processing modules at system level and duplication of E-units and F-units at a lower level. Furthermore, the VLSI I/O pin limitation problem is resolved through the use of wafer-scale integration for implementation of the processing modules. The intra-block communication takes advantage of the on-chip (on-wafer) interconnections, and the inter-block connections can have large bandwidths due to the increased size of the wafer.

In addition to utilizing some standard units, we have proposed a number of special-purpose VLSI components, such as a systolic multiplier [4] (as an example of an F-unit) and an associative memory module [5] (for use in the memory management operations).

The basic component of each E-unit is the elementary processor which performs simple logic and arithmetic operations. In order to accommodate such capabilities into an E-unit, we will take advantage of the OM2 Data Path ALU [6]. Each ALU bit occupies an area of .1mm \times .6mm. Considering the area needed for superbuffer drivers for the ALU control lines, the storage space for the controller, and the I/O ports, the total E-unit area for a 32-bit ALU is estimated at 4mm \times 4mm.

The *sub-net* consists of $(\log_2 n)-2$ stages of standard 2-by-1 arbitration switches followed by two stages of 2-by-2 square switches. The area of an arbitration switch, routing m bits in parallel, is about $30m\lambda \times 100\lambda$. Let n be the total number of the E-units, F-units, and the two input ports on a processing module. The length of the first stage of the network will then be $2000\lambda(n/2)$. The network requires $(\log_2 n)-2$ stages of arbitration and 2 stages of square switches, yielding a network width of $200\lambda(\log_2 n)$. With $n=32$, the network dimensions are $32000\lambda \times 1000\lambda$. The network delay for a token of 64 bits is $10(\log_2 n)-2+90$ ns if there is no conflict with the passage of the other tokens. With $n=32$, this delay is 120 ns.

The result of this analysis reveals that our proposal for implementation of the processing modules on silicon wafers is well within the limits of current technology. For example, with a conservative λ parameter (2.5 μ m), a system of 32 E-units can be easily implemented on a wafer. This size is sufficient enough to handle the majority of loops and small procedures. Naturally, as the λ size is shrunk due to the advances in technology, the number of E-units on a wafer can be increased. For example, with $\lambda=.8 \mu$ m, a 64-E-unit processing module will be reasonably practical.

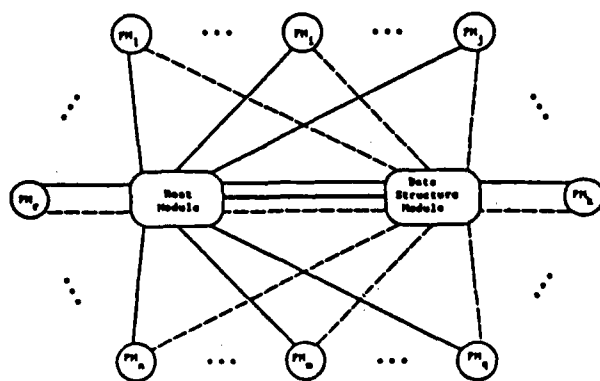
5. Conclusion

This paper has introduced a new multiprocessor system based on the data and block driven computation principles. The system has a cellular architecture suitable for VLSI implementation. The wafer-scale integration technology is used to speed up the intra-block communication, reduce the cost, and improve the I/O pin limitation problems. The system has been simulated both based on a probabilistic model and an emulation model. The simulation results cannot be

presented due to space limitations. The performance improves by increasing the number of E-units and processing modules, but saturates at some point due to I/O delays and single program environment simulation.

6. REFERENCES

1. Chang, T.L. and Fisher, P.D., "A Block-Driven Data Flow Processor," Proc. of the 1981 Int'l Conf. on Parallel Processing, Aug. 1981, pp. 151-155.
2. Shirazi, B., "WDDM- A Wafer-scale Data Driven Multiprocessor," Ph.D. Dissertation, University of Oklahoma, July 1985.
3. Arvind, Kathail, V., and Pingali, K., "A Processing Element for a Large Multiple Processor Dataflow Machine," 1980 Int'l Conf. on Circuits and Computers, Oct. 1980, pp. 601-605.
4. Hurson, A.R. and Shirazi, B., "A Class of Systolic Multiplier Units for VLSI Technology," Int'l Journal of Computer & Information Sciences, vol. 14, no. 5.
5. Hurson, A.R. and Shirazi, B., "A VLSI Design for the Parallel Finite State Automaton," ICCD '84, pp. 358-363.
6. Mead, C. and Conway, L., Introduction to VLSI Systems, Addison-Wesley, Mass., 1980.



PM_i = The i^{th} Processing Module.

Figure 1: The general organization of the proposed system.

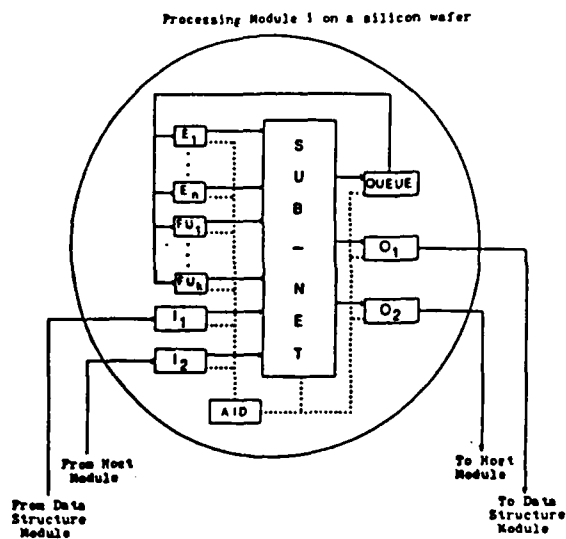


Figure 2: The overall organization of a processing module.

On Systolic Architectures for Interpolation and Integration

S.I.Omar and G.H.Masapati
Department Of Computer Science
University Of Ottawa
Ottawa,Canada K1N 9B4

Introduction

Recent advances in custom VLSI chip fabrication technology have made it economically feasible to implement parts of a system in hardware. A number of issues have to be addressed if this approach is used. One of them is the systematic derivation of a processing architecture from specification of subsystem. The paper addresses this issue. The basic approach used is the Transformational or Mapping approach. Our version of the approach differs from others [1-3] in several respects. After producing an Algorithm Model from Recurrence Relations we derive a Processing System Model algorithmically. Interpolation and Integration have been used as examples.

Algorithm Model

We propose a new graph model called Compute-Store Graph Model to describe computations being studied in this paper. Our formalization of an algorithm is as follows:

Definition 1. (Compute-Store Graph)

A Compute-Store graph is a labeled DAG(Directed Acyclic Graph) and is characterized by a 3-tuple (C, S, E) where

- (1) C is finite set of Computing nodes. Each computing node is assigned a unique co-ordinate in a 2-D Euclidean space. (Transformation 1)
- (2) S is a finite set of Storage nodes. With each storage nodes associated are two attributes (D, d) where D is the Data type and d is the delay. Each storage node is assigned a unique label from the set N of natural numbers. (Transformation 2)
- (3) E is a finite set of directed Eedges. Each incoming and outgoing edges are labeled uniquely from the set N .
- (4) $\forall w \in C, \exists u, v \in S : uw \in E \text{ and } wv \in E$
- (5) $\forall u, v \in C \text{ and } u \neq v, uv \notin E \text{ and } vu \notin E$
- (6) $\forall s \in S, \text{ there is at most one } u \in C : su \in E.$

Let $u \in C$, $A = \{w \in S : uw \in E\}$ and $R = \{w \in S : wu \in E\}.$

Definition 2. (Computable)

u is said to be Computable if arguments are available in elements(storage nodes) of A and elements(storage nodes) of R are empty.

Definition 3. (one clock tick)

Let t_r be the time taken to receive arguments from the elements of A . Let t_c be the time taken to compute the function by u . Let t_s be the time taken to store the results in the elements of R . Then t is said to be one clock tick if $t = t_r + t_c + t_s$.

Definition 4. (Rules of Computing)

- (1) u is computable.
- (2) u receives arguments from elements(storage nodes) of A , computes the function, and stores the result in elements(storage nodes) of R , in one clock tick.

Model of Processing System

The basic objects of the model are: Processing Elements(PE) and Communication Links(CL). PE performs the necessary functional transformation using the data communicated to it via CL through input ports and data in its storage registers. It performs the functional transformation after the elapse of specified units of time contained in its status register. The delay in the activation of functional transformation ensures that all the required data are available. After the transformation the resulting

data are communicated to other PE via CL through output ports. CL provides the facility for inter-PE communication. Each CL starts at an output port and terminates at an input port. It communicates data of specified type.

In terms of these basic objects a Processing System(PS) is defined as a 2-tuple (PE,CL), where PE is a finite set of Processing Elements and CL is a finite set of Communication Links. PE is defined as a 5-tuple (FT,IP,OP,ST,SR), where FT is a finite set of functional Transform Units, IP is a finite set of Input Ports, OP is a finite set of Output Ports, ST is a finite set of Status Registers and SR is a finite set of Storage Registers. CL is defined as a 3-tuple (OP,IP,EV), where OP is a finite set of Output Ports, IP is a finite set of Input Ports and EV is a finite set of Data Types. Also $IP = IPI \cup IPE$ where IPI is a finite subset of Input Ports at the termination of CL and IPE is a finite subset of Input ports from the Environment to PS. Similarly $OP = OPI \cup OPE$.

Processing Architectures for Interpolation and Integration

Recurrence relation for Neville's Interpolation is :

$$Q_{i,0} = f(x_i) ; i = 0,1,\dots,n$$

$$Q_{i,j} = [(x-x_i) * Q_{i-1,j-1} - (x-x_{i-j}) * Q_{i,j-1}] / (x_{i-j} - x_i) ; i = 1,2,\dots,n \text{ and } j = 1,2,\dots,i$$

The Compute-Store graph for this relation and the corresponding Processing Architecture are shown in Figs. 1 and 3.

Recurrence relations for Romberg Integration are :

$$R_{1,1} = 0.5 * (b-a) * [f(a) + f(b)]$$

$$R_{k,1} = 0.5 * [R_{k-1,1} + h_{k-1} * \text{SUM} \{ f(a + (i-0.5) * h_{k-1}) \} \text{ from } i = 1 \text{ to } 2^{*(k-2)}] ; k = 2,3,\dots,n$$

$$h_k = b-a / 2^{*(k-1)}$$

$$R_{i,j} = [R_{i-1,j-1} - (4^{*(j-1)}) * R_{i,j-1}] / (1 - (4^{*(j-1)})) ; i = 2,3,\dots,n \text{ and } j = 2,3,\dots,i$$

The Compute-Store graph and Processing Architecture for this relation are shown in Figs. 2 and 4.

Mapping Algorithm

The mapping process consists of applying the following transformations between the basic objects of Algorithm Model and Processing System Model.

Transformation 1: Define $T1 : C \rightarrow E_s^2$ s.t. $T1(C_{ij}) = (i,j)$

Transformation 2: Define $T2 : S \rightarrow N$ s.t. $T2(s_i) = i$ where $i \in N$

Transformation 3: Define $PE = T3.C$ where $T3$ is an $n \times n$ integer matrix. One feasible way of choosing $T3$ is : $T3 = I$

Transformation 4: Define $T4 : (S,E) \rightarrow (SR,ST)$. Algorithm A constructs $T4$.

Transformation 5: Define $T5 : (C,S,E) \rightarrow CL$. Algorithm B constructs $T5$.

Transformation 6: Define $T6 : (S,E) \rightarrow (IP,OP)$. Algorithm C constructs $T6$.

Algorithms A,B and C have been described in [4].

Conclusion

We have shown that by representing a Recurrence Relation as a Compute-Store Graph it is possible to algorithmically obtain a feasible Processing Architecture in the form of Processor-CommLink Model. This architecture is solely based on the characteristics of the Recurrence Relation. No external constraints are imposed on it. Our Mapping Algorithm produces a speed-up of $O(n)$ for the example architectures. Several related issues are being investigated. One of them is obtaining an optimal from the feasible architecture.

References

- [1] D.I.Moldovan and J.A.B.Fortes, "Partitioning and Mapping Algorithms into Fixed Size Systolic Array", IEEE Tr. Comp., C-35(1), Jan. 1986, 1-12.
- [2] I.V.Ramakrishnan, D.S.Fussel and A.Silberschatz, "Mapping Homogeneous Graphs on Linear Arrays", IEEE Tr. Comp., C-35(3), Mar. 1986, 189-209.
- [3] I.Koren and G.M.Silberman, "A Direct Mapping of Algorithms onto VLSI Processing Arrays", Proc. 1983 IEEE Intl. Conference on Parallel Processing, 335-337.
- [4] S.I.Omar and G.H.Masapati, "On Mapping of Algorithms onto Processing Systems", (to be submitted for publication).

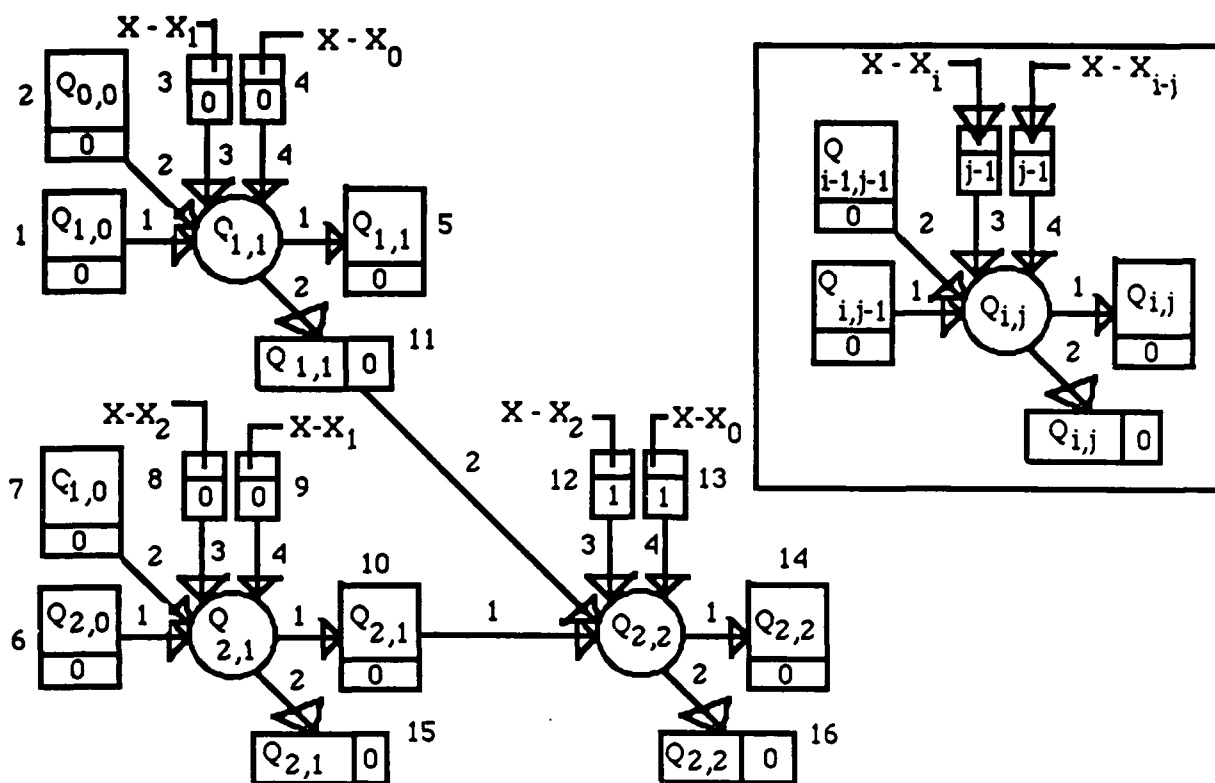


Fig. 1. Compute-Store graph for Neville's Interpolation

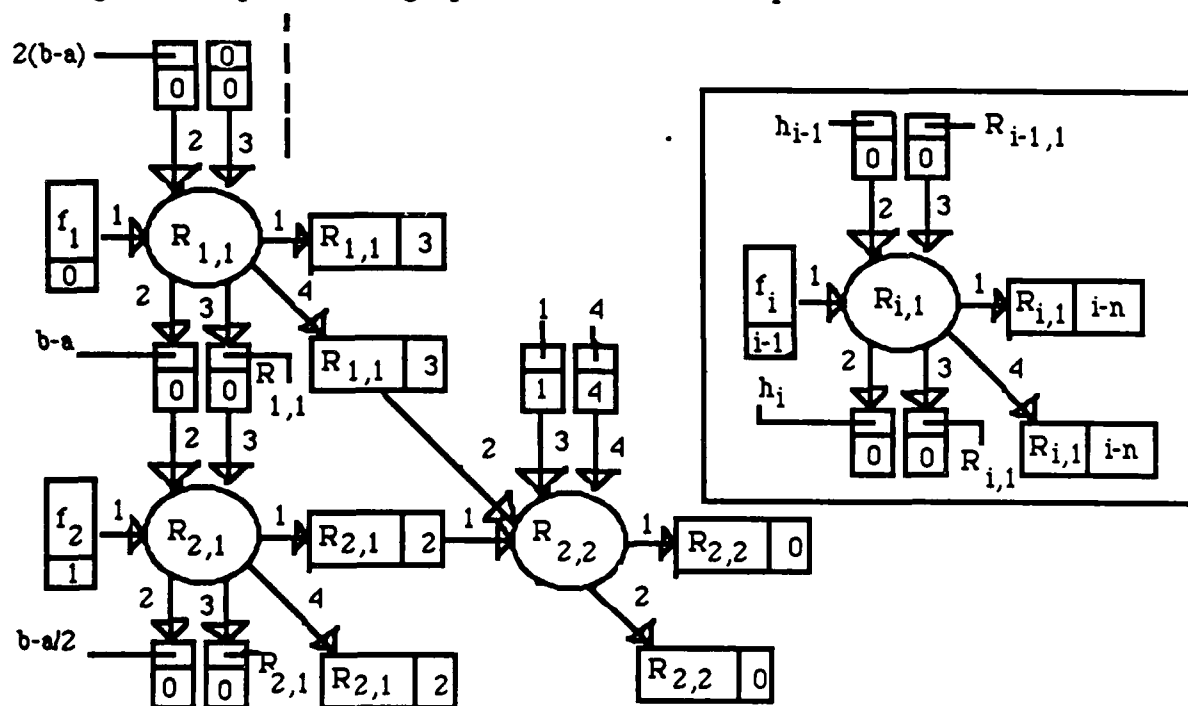


Fig. 2. Compute-Store graph for Romberg Integration

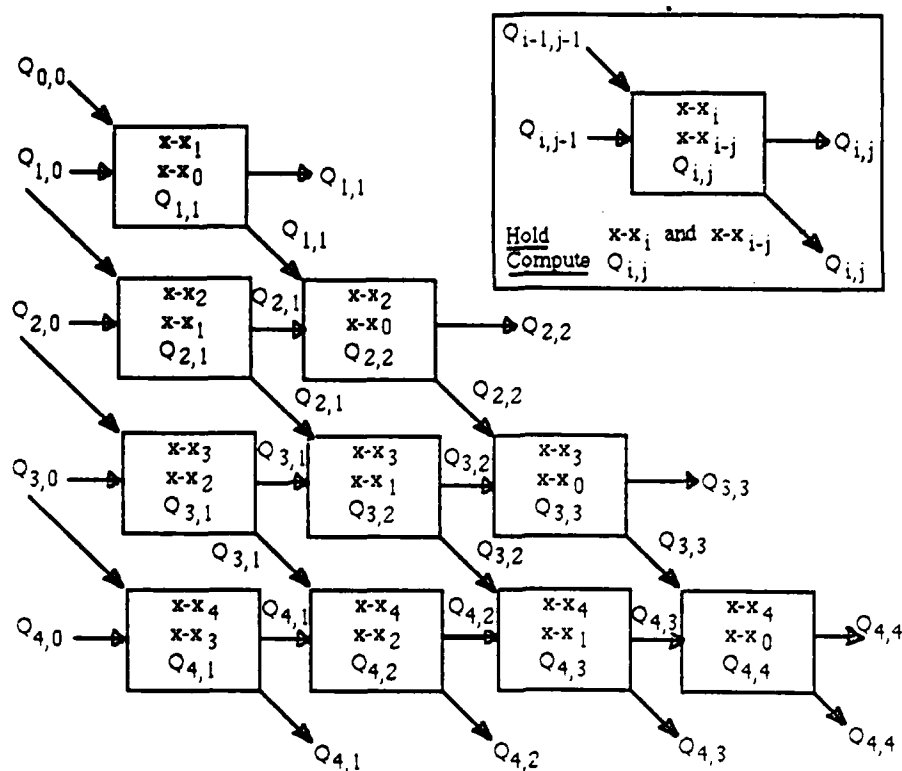


Fig. 3 Processing Architecture for Neville's Interpolation

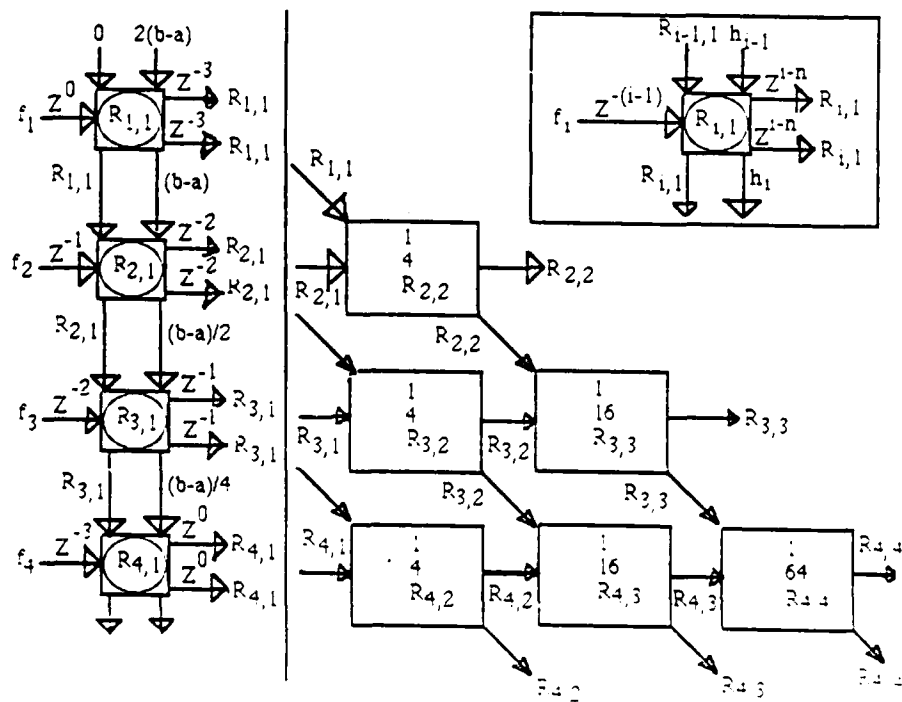


Fig. 4 Processing Architecture for Romberg Integration

An Applicative Programmer's Approach to Matrix Algebra, Lessons for Hardware and Software

David S. Wise dswise.indiana@csnet-relay
Computer Science Department
Indiana University
Bloomington, IN 47405-4101

It has now been ten years since the "data pull" nature of lazy applicative programming was proposed as a solution to the process fragmentation problem of parallel processing [3]. Since that time there has been considerable development of hardware, both pipelined/vector machines and, more recently, cube- and switch-connected engines composed of tens and hundreds of processors. The style of programming for the former machines has not improved during that time, and we yet have no languages suitable for taking appropriate advantage of the latter.

At the same time, the early development of machines derived from applicative languages has proceeded from combinator-based hardware to more elaborate multiprocessor architectures. **This paper proposes** algorithms derived using applicative programming, suitable for **multiprocessors switch-connected to a banked (self-managing) heap memory**. Although shared-memory access to M locations *requires* time at least logarithmic in M , we are accustomed to constant-time access (for small M —or cache), which is finally disappearing in tree- and switch-connected multiprocessor proposals. Moreover, there are several solutions, some in hardware [15], for distributing management of a shared heap.

The Daisy Project at Indiana has been exploring parallelism through implementation of and practice in an applicative (or functional) programming language. One facet of this effort is extending this style, well grounded as it is in formal semantics, back toward hardware in order to discover the essence of a rich run-time environment, suitable to achieve the efficiency apparent in the high-level program. Aiming at a well known problem area, Matrix Algebra, this overview demonstrates the power of applicative programming in impressing a new, parallel perspective on a well studied problem.

Premises:

Applicative (functional) programming is a programming style [3, 11] in which the only control structure is the application of function to argument (-list). There is no rebinding of identifiers (assignment statement), except through binding of parameters to arguments, and no sequential flow except that inferred from primitive functions (such as addition where the values of all terms are necessary before the computation of their sum.) Functional operations, like composition and mapping, should allow for one function application to return several results, just as it may take several arguments [4].

A significant feature of this style is that there is no inherent sequentiality of execution. Relaxed (lazy) evaluation order means that arbitrary parallel evaluation is safe. This fact makes Daisy most suitable both for describing parallel algorithms and for describing parallel hardware [7].

Once we have the ability to identify parallelism within a large function (program), there remains the problem of which processes to run with only finitely many processors. While the problem is unsolved in general, we observe that it is better to dispatch processes as high as possible in the structure of the program. This "height heuristic" is an effort to create long-running processes in order that the overhead of process dispatch and recovery be amortized over as long a parallel computation as possible. If processes are only dispatched from points deep in the structure of a program (e.g. only in FORTRAN's arithmetic expressions) then they will be too short-lived to justify their creation. Therefore, it is important to lift parallel-processing notation to as high a level as possible, providing it throughout the programming language, if possible.

Application to Matrices:

If applicative programming is a better style for driving parallel architectures than conventional/iterative languages (like FORTRAN), then it ought to be able to offer some new perspective—or even an improvement—on problems for which those languages are thought to be suited. As a test of this premise, we consider important problems from Matrix Algebra, where pipelined/vector processors ought to be optimal.

The following is a thumbnail sketch of an applicative approach to matrix algebra. It is remarkable because it demonstrates that the selection of a tree-like data structure, directly suggested by the recursive paradigm and contrary to most memory architecture, leads to new parallelism, a new strategy for processor allocation, and an efficiently stable algorithm.

Representation is a most important issue. A conventional structure in the highly recursive style of functional languages is the tree. Thus, we are drawn to finite trees for representing finite structures: specifically, an n -dimensional array will be represented by a 2^n -ary tree, as in Figure 1. The exact size of an array is implied by data independent of this structure, because a scalar is to be interpreted as such an array of *any* size. That is, the scalar, 3, may be interpreted as the (2-dimensional) matrix of arbitrary size that is three times the identity matrix; the identity matrix is 1 and 0 is the zero matrix. A matrix is either such a scalar or it is four quadrants, each a (sub)matrix.

Implicit in this definition is that sparse matrices are represented sparsely and, therefore, that the space necessary to fill out this Strassen-like decomposition [10] to a large matrix of size $2^m \times 2^m$ is negligible. The preferred memory architecture, however, is a heap.

Figure 2 shows the process decomposition for conventional operations like matrix addition and multiplication. It exhibits two facts: first that task decomposition occurs at the root, into four or sixteen large subprocesses, which will run for quite a while amortizing the expense of their dispatch/recovery; quite an improvement over parallelism only at the leaves of an expression tree! Second is behavior on uncovering sparseness; sparse behavior is effected when the algorithm encounters a scalar 0 instead of a pointer to a large matrix. When an argument to addition or multiplication includes a 0 (or 1) quadrant, then 25% of the effort is annihilated by borrowing a reference to pieces of the other operand [13, 14].

More important, however, is matrix inversion, or solving bilinear forms. Figure 3 illustrates the decomposition for a Pivot Step at a known position pivot position. The solution is to return the pivoted matrix, and the pivot row and column, each represented as a binary tree. The problem decomposes four ways into quadrants identified as that containing the (non-zero) pivot element (*piv*), those coordinating horizontally (*row*) and vertically (*col*), and that diagonal (*off*). The algorithm is interesting [16], admitting massive parallelism in the transformations of *row*, *col*, and *off*.

The possibility of performing full row/column pivoting is implicit with essentially *no* increase in search cost. Suppose that before beginning the pivoting the tree structure had been recursively traversed and annotated at each nonscalar subtree with the largest contained (absolute) value and with two bits indicating in which quadrant it lay. Sufficient information exists at the root of the tree to initiate the first pivot step without further search. Moreover, those maxima may be maintained by the Pivot Step transformation, because the information is local to values/subtrees being changed, and because that maxima information is static *off* subtrees unaltered (because they coordinated on zeroes in the pivot row or column) by the transformation. Thus, this sufficient information will be maintained at the root of the tree to initiate *each successive* Pivot Steps without further search.

Acknowledgement: Research reported herein was supported by the National Science Foundation under a grant numbered DCR 84-05241.

References

1. S. K. Abdali & D. D. Saunders. Transitive closure and related semiring properties via eliminants. *Theoretical Computer Science* 40, 2,3 (1985), 257-274.

**Session 15C: Applicative Language and
Data Flow Techniques**

Chairperson: Kim Gostelow
GE Research and Development Center

IMPLEMENTING LOGICAL VARIABLES ON A GRAPH REDUCTION ARCHITECTURE¹

Gary Lindstrom

Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

1. Functional and Logic Programming

Logical variables offer a semantic meeting ground between functional and logic programming languages [7]. From the perspective of functional programming, logical variables present a radically different basis for information flow. In particular,

- * logical variables are not directly bound by uniquely determined expressions, in contrast to the customary dataflow orientation of functional languages;
- * rather, they denote values determined by the intersection of successively applied constraints.

In a functional programming language with logical variables, this difference can permit novel effects otherwise quite difficult to obtain. These include:

- * an *action at a distance* effect, whereby certain kinds of "side-effects" can be achieved within an applicative setting (the Prolog "difference list" trick for constant time list concatenation is a familiar example);
- * a stronger *object orientation*, under which logical variables play the role of mutable shared entities, and
- * the use of this object orientation as a basis for *isotropic* process communication and synchronization, as in Concurrent Prolog [8].

2. Reduction Architectures

Graph reduction offers many advantages as a strategy for executing applicative programs. These include:

- * very flexible patterns of concurrency and load distribution on parallel architectures;
- * clean support for infinite and cyclic data structures through demand evaluation [1];

¹This material is based upon work supported by NSF Grant DCR 8506000, and a Shared University Research Grant from the IBM Corporation. A full paper on this subject is available from the author.

- * good program and data locality effects;
- * the support of recursion and iteration (via tail recursion) without the need for explicit token objects or "color" tags and associative memories for their assembly into operator "firing sets", and
- * the ability to incorporate a form of data-drive as a submode.

The principal prices to be paid for these advantages are potentially costly storage management, and the added overhead of demand propagation. A good survey of reduction architectures may be found in [9].

3. FGL+LV on Rediflow

We have developed a strategy for implementing FGL+LV [5], a functional language with logical variables, on the Rediflow multiprocessing graph reduction architecture [2]. The aspects of logical variables receiving special consideration include:

- a. parallel unification, especially proper treatment of indeterminate behavior, e.g. mutual exclusion on variable binding [3];
- b. variable binding through emulated graph node merging;
- c. exploitation of two levels of demand: *assertive* (during unification), and *non-assertive* (ordinary "read-only" usage), and
- d. avoidance of meaningless cyclic variable bindings.

The existing base language implementation is smoothly extended, with a word size increase of only one bit (needed to implement two levels of demand). Lazy evaluation in the base language is retained, except that actual parameters are now made strict to one level of evaluation. This requirement, overlooked in [5], is argued to be semantically and operationally inescapable in a functional language with logical variables. It also provides insight into the vexing problem of how to apply the occur check in a language with infinite data objects. A novel technique for merging cyclic lists is used to implement logical variable binding in a distributed manner without locking or busy waiting.

4. Extensions

Our work thus far has focused on the adoption of logical variables in a deterministic setting, i.e. without support for backtracking or OR-parallel search [4]. This single extension brings into the realm of functional programming such elegant techniques as the Milner algorithm for polymorphic type checking [6]. And, while the absence of OR-parallelism may be lamentable, in partial compensation we retain functional programming's execution *directionality*, which offers a clean solution to the AND-parallel control problem which plagues pure logic programming.

Continuation work is underway aimed at augmenting this implementation approach to embrace OR-parallelism.

References

- [1] R.M. Keller and G. Lindstrom.
Applications of feedback in functional programming.
In *Conference on functional languages and computer architecture*, pages 123-130.
October, 1981.
- [2] R.M. Keller, F.C.H. Lin, and J. Tanaka.
Rediflow multiprocessing.
In *IEEE Compcon '84*, pages 410-417. Feb., 1984.
- [3] G. Lindstrom.
OR-parallelism on applicative architectures.
In Sten-Ake Tarnlund (editor), *Proc. Second International Logic Programming Conference*, pages 159-170. Uppsala University, July, 1984.
- [4] G. Lindstrom and P. Panangaden.
Stream-based execution of logic programs.
In *Proc. 1984 Int'l. Symp. on Logic Programming*, pages 168-176. February, 1984.
- [5] G. Lindstrom.
Functional programming and the logical variable.
In *Symposium on Principles of Programming Languages*, pages 266-280. ACM,
January, 1985.
- [6] R. Milner.
A theory of type polymorphism.
J. of Comp. and Sys. Sci. 17(3):348-375, 1978.
- [7] U.S. Reddy.
On the relationship between functional and logic languages.
In D. DeGroot and G. Lindstrom (editors), *Logic Programming: Functions, Relations, and Equations*. Prentice Hall, 1986.
To appear.
- [8] E.Y. Shapiro.
A Subset of Concurrent Prolog and Its Interpreter.
Technical Report TR-003, Institute for New Generation Computer Technology,
January, 1983.
- [9] P.C. Treleaven, D.R. Brownbridge, and R.P. Hopkins
Data-driven and demand-driven computer architecture.
Computing Surveys 14(1):93-143, March, 1982.

MULTI-PROCESSOR REDUCTION MACHINES

J.L. Meador and M.L. Manwaring

Washington State University Electrical and Computer Engineering
Pullman, Washington 99164-2210 (509)335-6602

For future symbolic multi-processing, networks of several hundred processors are anticipated, each consisting of 10K-20K devices. At 1984 commercial densities this equates to 4 to 8 devices per chip. According to anticipated 1988 densities [1], this can be expected to increase to as many as 128 per chip. We are quickly approaching a time when the technology will support large scale symbolic parallelism, but only if we can find ways to efficiently implement it. To help achieve this goal, there is a need to investigate moderately sized, computationally versatile processors designed for efficient inter-processor communication within a multi-processor organization.

There exists a general trend at present toward language directed architectures. RISCs, CISCs, and direct execution machines have all attacked the uni-processor design problem with various language directed goals.

RISC architectures are designed with the goal of CPU control simplification. This simplification can be made based upon instruction frequencies in compiled languages. Here, more frequent instructions are implemented with fewer clock cycles. A simpler control section also allows for additional on-chip register space. CISC architectures take a contrary approach, that of CPU control enhancement. This enhancement is less based upon frequencies of common instructions than it is upon building new ones. These new instructions are intended to accomplish in one main memory access what would have otherwise required several. Both RISC and CISC approaches share the goal of providing more efficient support of compiled high level languages. Whether one does so better than another is a subject of debate. It is not the intention of this paper to take a stand on that issue. The point here is that both approaches share a design philosophy based upon some aspect of abstract programming language semantics.

Direct execution processors discard the conventional RISC and CISC machine instruction set notion for the direct representation of a high level language. Once again, the idea is to construct hardware specifically suited to an abstract programming language. High performance Lisp machines have been commercially available for some time now. At least eight commercial development efforts for machines directly executing Ada, Prolog, Forth, Lisp and other languages are under way. Impressive performance measurements have been reported [3].

Functional programming architectures represent one language directed approach to multi-processing. Functional languages are a subclass of declarative programming languages, of which Prolog is perhaps the best known member. Declarative languages provide flexible expression of general purpose algorithms, while implicitly expressing parallelism. The interested reader may find a useful development on functional languages in [4]. Informative surveys of architectures influenced by this programming paradigm may be found in [6] and [8].

Several aspects affect the design of a multiprocessor for functional programming languages. These include the language itself, the execution algorithm, and the hardware organization.

To make effective use of current and anticipated technology, it is felt that the selected machine language must have a simple basis, be expressive, and be extendable. If the language starts out simple, then less hardware area will be expended to implement the bare essentials. Furthermore, the language will not be useful in a general purpose sense if it is not expressive and extendable. Berkling's lambda-reduction language [2] and Turner's combinators [7] have been considered by the authors for kernel languages. Lambda reduction languages are composed of lambda

calculus expressions having string based reduction semantics. Turner's combinator language is similar, except that it has graph based reduction semantics, and makes use of special purpose functions called combinators. Berkling's lambda reduction language and its derivatives have the advantage of being more readable, at the cost of the extra complexity due to bound variables. Combinators, on the other hand, sacrifice readability for a simpler implementation requiring no bound variables. Lambda-reduction languages are the current focus of the authors' work because of their similarity to Lisp. Combinators could play an important future role if they promise superior multiprocessing performance or hardware simplification is required. It is expected that once more is understood about implementing these two languages, then others such as functional Lisp dialects, FFP, and Hope may be considered.

The execution algorithm specifies hardware state transitions in terms of machine language syntax and semantics. Internal representation, and the interpretation method each play a role in the determination of an execution algorithm.

Internal representation is the way machine language is encoded to represent part of the hardware state. Virtually all computing machines use either a string based representation or one that is graph based. Conventional instruction sets use a combination of the two. In string based systems, logically adjacent tokens are physically adjacent in machine memory. In graph based systems, logically adjacent tokens are linked by indirect references using physical addresses, so need not be physically adjacent. String based program representations are widely recognized to have an undesirable difficulty with sequential evaluators. To update a reference in a string representation, all occurrences of that reference within the string must be re-written. This can require scanning the entire program string -- clearly an undesirable action in a sequential evaluator. Graph representations avoid this problem by allowing each reference to point to the same value, so only one update need be performed to satisfy all references. By exploiting this property, graph representations eliminate the need to re-evaluate sub-expressions, and are more compact. These are all useful attributes for sequential evaluation. However, complex memory management requirements (including garbage collection), and the "fine-grained violation" of the Church-Rosser property make graph based representations less desirable in a multi-processor. String representations have a predictable information flow, and ability to take advantage of simple memory management schemes. For these reasons, it is felt that string or at least predominantly string based representations should be considered for multi-processor implementations.

Closely related to internal representation is the issue of how a given representation is interpreted. Interpretation by evaluation implies program values are combined to produce a result. Interpretation by reduction implies that the program string is repetitively re-written to obtain a result. It is felt that reduction offers a better alternative for a multi-processor because state and control information can flow as a unified stream between processing elements.

Several hardware organizations are under consideration for reduction based multiprocessing. A simple linear organization, for example, is one under investigation. A processing element of this array uses string reduction. It accepts input tokens, expecting a left-to-right traversal of the program expression's internal representation. At any moment, it is responsible for the reduction of a single sub-expression.

Since the current target is Berkling's lambda reduction language, the program string consists of a sequence of atoms and constructors. Constructors combine expressions to form larger ones, in a way similar to what CONS does for lists in Lisp. Constructors also act as special operators in the languages. The two most important constructors are lambda and apply.

If a lambda constructor is encountered, then a beta-reduction based on the beta-conversion operator of lambda calculus is performed. If an application constructor (applicator) is recognized, tokens are absorbed until either another constructor is encountered, or the end of the reducible expression is reached. If another constructor is encountered after the applicator, all tokens absorbed to that

point are passed to the output, that information becomes available to a neighbor processing element, and interpretation proceeds with the new constructor. If no additional constructors are found, that means the innermost expression has been reached and a function application is performed. Lambda constructors follow an outermost reduction rule, and applicators follow an innermost rule. The innermost function applications are recognized because the processor only remembers information since the last applicator. Only outermost lambda reductions are performed by an element because sub-constructors are ignored. For a single traversal of the program representation, either all innermost function applications or all outermost lambda expressions will be reduced.

A linear array of such processing elements yields a network similar to the Newcastle reduction machine [5]. It differs in that the program expression flows in one direction, helping simplify the processing element and inter-processor communications. It is expected that a processing element having a useful set of functional primitives will require 10K to 20K devices for control and local memory. As a result, multi-processor reduction machines based upon these ideas are expected to offer a viable approach for future symbolic processing.

REFERENCES

- [1] -, "Report of the Research Briefing Panel on Computer Architecture", Research Briefings 1984 for the Office of Science and Technology Policy, the National Science Foundation, and Selected Federal Departments and Agencies, National Academy Press, 1984.
- [2] K.J. Berkling, "Reduction languages for reduction machines", Proc. 2nd Int. Symp. Computer Architecture, IEEE, New York, 1975.
- [3] C.C. Cole, "A Pride Of New CPUs Runs High-Level Languages", Electronics, November 25, 1985.
- [4] H. Glaser, C. Hankin, and D. Till, Principles of Functional Programming. Prentice-Hall International, 1984.
- [5] P.C. Treleaven and G.F. Mole, "A multi-processor reduction machine for user-defined reduction languages," Proc. 7th Int. Symp. Computer Architecture, IEEE, New York, 1980.
- [6] P.C. Treleaven, et.al. "Data-Driven and Demand-Driven Computer Architecture" ACM Computing Surveys, Vol. 14, No. 1, March 1982.
- [7] D.A. Turner, "A New Implementation Technique for Applicative Languages", Software Practice and Experience, January 1979.
- [8] S.R. Vegdahl, "A Survey of Proposed Architectures for the Execution of Functional Languages", IEEE TRANS. ON COMPUTERS, Vol. C-33, No. 12, December 1984.

Massive Fine-Grain Parallelism in Array Computation - a Data Flow Solution

Guang R. Gao
Department of EECS, M.I.T.
Cambridge MA 02139

Abstract

This paper argues that massive parallelism in vector computation can be exploited effectively utilizing data flow principles in future generation computers. The key is to organize the data flow machine program graph such that array operations can be fully pipelined. The applicative nature of the data flow graph model allows flexible scheduling of the execution of enabled instructions in the pipeline data flow programs. Accordingly, program transformation can be performed on the basis of both the global and local data flow analysis to generate efficient pipelined data flow machine code. A pipelined code mapping scheme for transforming array operations in high-level language programs into pipelined data flow machine programs is described.

1 Massive Parallelism in Array Computation

A major driving force in the development of high-performance computers has been scientific computation. The kernels of such computations typically are expressed in linear algebra with all data structured as elements of arrays. In the computation, the bulk of the elements of an array are processed in a regular and repetitive pattern through the different phases of execution. For example, many applications take the form of computing successive states of a system represented by physical quantities on an Euclidean grid in two or three dimensions, and the new values of each grid point may be computed independently. Thus, the degree of concurrency is often at least equal the number of the grid points (for a $100 \times 100 \times 100$ case, the parallelism will be well over a million!). Therefore, the efficient mapping of the massive parallelism of array computation into machine-level code structure has been a major consideration in the design of high-performance computer architecture as well as its program transforming compilers.

2 Fine-grain Parallelism and Array Computation

2.1 Pipelining of Data Flow Programs

Fine-grain parallelism exists in two forms in a data flow machine level program, as shown in Figure 1, which consists of seven actors divided into four stages. In Figure 1 (a), actors 1 and 2 are enabled by the presence of tokens on their input arcs, and thus can be executed in parallel. Parallelism also exists between actors 3 and 4, and between actors 5 and 6. In static data flow architecture, we can arrange the machine code such that successive computations can follow each other through one copy of the code. If we present a sequence

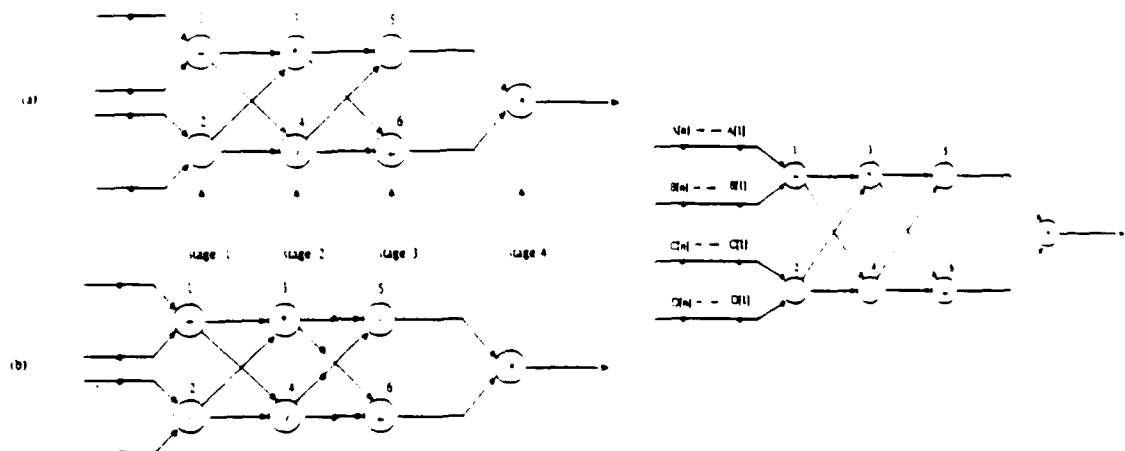


Figure 1 Pipelining of Data Flow Programs

of values to the inputs of the data flow graph, these values can flow through the program in a pipelined fashion. In the configuration of Figure 1 (b), two sets of tokens are pipelined through the graph, and the actors in stage 1 and 3 are enabled and can be executed concurrently. Thus, both forms of parallelism are fully exploited in the graph.

2.2 Data Flow Languages and Array Computations

The use of data flow languages such as Val [1] encourages an applicative style of programming which does not depend on the von Neumann style of machine program execution. The basic operations of the language, including operations on arrays, are simple functions that map operands to results. Data dependencies, even those involving arrays, should be apparent. Since our major concern is how to utilize the regularity of array operations in the source program, we concentrate on two array creation constructs – the forall and for-construct expressions.

The forall construct allows the user to specify the construction of an array where similar independent computations can be performed on all elements of the array. Figure 2 (a) is an expression which defines a one-dimensional array X from an input array A. The for-construct expression, proposed as a special case of the Val iteration construct, is used to specify construction of an array where certain forms of data dependencies exist between its elements. Figure 2 (b) is a for-construct expression which constructs an array X based on the first-order linear recurrence, using array A and B as parameter arrays.

The two constructs provide a means to express array construction operations of desired regularity without using array append operations. Expressions based on these constructs are the major code blocks studied in this paper.

```
X : array(real) :=
forall i in [0, m+1]
construct
  if i = 0 then A[i]
  elseif i = m+1 then A[i]
  else
    (A[i-1] + A[i] + A[i+1])/3
  endif
endforall
```

(a)

```
X = array(real) =
for i from 0 to m-1
  T = array(real) from array-empty
  construct
    if i = 1 then x0
    else A[i]*T[i-1] + B[i]
  endif
endfor
```

(b)

3 Pipelining of Array Operations

One objective of the machine code mapping scheme for static data flow computers is to generate code which can be executed in a pipelined fashion with high throughput. The pipeline must be kept busy; computation should be balanced and no branch in the pipe permitted to block the data flow. Furthermore, computation resources should be efficiently utilized. In particular, the usage of storage for arrays is important because the user program usually contains vast amounts of array data to be processed.

In a data flow computation model, an array value can be regarded as a sequence of element values carried by tokens transmitted on a single data flow arc. In Figure 1 (c), the four input arcs are presented with four input arrays A, B, C, D; all are spread in time. Thus, the graph can be *fully pipelined*.

We can observe that each actor in Figure (b) is performing a vector operation, e.g., actor 1 – vector addition, actor 2 – vector subtraction, etc., a total of seven vector operations. However, unlike the vector operations usually supported in conventional vector processors, there is no requirement that the activities of one such vector operation be continuously processed by one or a group of dedicated function units in the processor. The applicative nature of the data flow graph model allows flexible scheduling of the execution of enabled actors in the pipeline. In fact, an ideal data flow scheduler (with a sufficiently large data flow computer) will execute each actor as soon as its input data become available. As a result, the activities of the seven vector operations overlap each other, performing operations on different elements of different arrays concurrently. Therefore, massive parallelism of vector operations can be effectively exploited by a data flow computer in a fine-grain manner: the scheduling of the physical function units and other resources for sustaining such vector operations are totally transparent to the user.

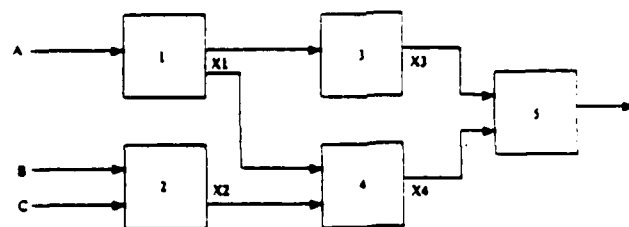


Figure 3: A group of code blocks

4 Program Mapping and Optimization Techniques

We have developed a pipelined code mapping scheme which concentrates on the analysis and handling of the two types of code blocks [6]. It is essentially a two-step process. The first step consists of the application of a set of basic mapping rules which can translate the code blocks into pipelined data flow graphs. In this step, the conceptual model of arrays in the source program - i.e. the input and output arrays as seen by each code block - remains unchanged, but the array operations are translated into corresponding data flow actors in the result graph. The second step consists of the application of a set of optimization procedures which can remove the array actors from the result graphs of step 1 and replace them with ordinary graph actors. Thus, the links between code blocks become ordinary arcs of a data flow graph. The result graph for a pair of producer and consumer code blocks may be executed concurrently, both in a pipelined fashion, without involving array operations.

5 Summary

Based on the pipelined code mapping strategy, the performance analysis of a recently proposed data flow supercomputer architecture consisting of 256 processors has reported promising expectation in a number of benchmark problems, notably the global weather simulation model [3], the three dimensional aerodynamic simulation problem [4], and several benchmarks studied in [7]. It remains to be seen how broadly applicable are such code mapping scheme in data flow supercomputing.

REFERENCES

1. Ackerman, W. B. and J. B. Dennis. "Valj' A Value-Oriented Algorithmic Language Preliminary Reference Manual.", Technical Report 218, Laboratory for Computer Science, MIT, Cambridge, MA, 13 June 1979.
2. Ackerman, W. B., "Data Flow Languages", AFIPS Proceedings, Vol. 48: Proceedings of the 1979 National Computer Conference, AFIPS, 1979.
3. Dennis, J. B., Gao, G. R. and Todd, K. W., "Modeling the Weather with a Data Flow Supercomputer", IEEE. Trans. on Computers, C-33, 7, July, 1983.
4. Dennis, J. B., "Data Flow Computation - A Case Study", submitted for publication, Lab. for Computer Science, MIT, Cambridge, MA, Jan. 1986.
5. Gao, G. R., "A Maximally Pipelined Tridiagonal Linear Equation Solver", to appear on the Proceeding of IFIP Highly Parallel Computer Conference, Nice, France, March 1986.
6. Gao, G. R. "A Pipelined Code Mapping Scheme for Static Data Flow Computer". Ph.D dissertation in preparation, Lab. for Computer Science, 1986.
7. Adams, G. B., Brown, R. L. and Denning, P. J., "Report on an Evaluation Study of Data Flow Computation", Research Institute of Advanced Computer Science, April, 1985.

AUTOMATED DATA FLOW DIAGRAM VERIFICATION

Reva Friedman
Northwestern University, Evanston, Illinois

Waldo C. Kabat ^{*}
University of Illinois at Chicago, Dep. of Electrical Engineering
and Computer Science

Wojtek Kozaczynski
University of Illinois at Chicago, Dep. of Information and Decision
Sciences

SUMMARY

An initial implementation of an interactive Automated Data Flow Diagram Verification system (ADFDV) is described. ADFDV is a component of the larger system called System Analyst's Apprentice (SAA). SAA is being developed to assist an analyst in different phases of Computer Information System (CIS) design and development. The ADFDV is a tool for mechanical checking of Data Flow Diagrams for correctness. The philosophy behind the system is to assist the analyst in the error prone job of the verification of Data Flow Diagrams for large programming projects.

Structured Analysis and Design techniques have been used to avoid errors that may have an impact on the quality of final CIS design. First, in early 1970s, these techniques were successfully applied to the development of highly structured programs. In late 1970s Edward Yordon and Larry Constantine first described a structured process-oriented method for system analysis and design. This method is based on functional system decomposition and stresses flow and transformation of data in the computer system. It uses diagramming techniques for system design specification. Data Flow Diagrams are the focal point of those techniques. They are used as a formal descriptive language as well as informal means of communication between system developers and future system users. They are used to describe the system on very high logical level as well as on the low physical level.

The system analysis and design method first proposed by Yordon and

* Responsible for all correspondence

Constantine and later improved by others is well known under the name of Data Flow Analysis Method (DFAM) and has been widely accepted by the IS professionals. The purpose of the present work is to develop a prototype expert system, System Analyst's Apprentice (SAA), to assist the analysts in the process of application design and development through DFAM techniques. The greatest advantage of the system comes from the fact that it will free the analyst from a great amount of tedious and low level work. That in turn will give him the opportunity to direct his attention to higher level problems.

The current system implementation is based on the formalization of the primitives for the interactive design of DFDs and the rules for DFD verification using knowledge representation language MRS as a tool. New primitives and new rules can be added to the system. The greatest advantage of ADFDV can be realized when interactive development of a Data Flow Diagram is done with the verification option running in the background. The design and implementation of ADFDV serves as a testbed for the design of System Analyst's Apprentice.

Keywords: Expert Systems, Information Systems Analysis, Information Systems Design, Data Flow Analysis, Data Flow Diagram Verification.

FINELY GRAINED PARALLELISM IN AN APPLICATIVE ARCHITECTURE

John T. O'Donnell
Computer Science Department
Indiana University
Bloomington, Indiana 47405

THE APSA PROJECT

The Applicative Programming System Architecture (APSA) research project [2, 3, 4, 5] is developing new ways to use finely grained parallel architectures to support applicative programming languages. The goal is a parallel machine that is easily programmable with a LISP-like language. The programmer does not need to control the parallelism explicitly. Instead, the system provides a number of rich types of data structure with parallel operations that perform complex operations on the data structures. Some of these data structures are used by the language interpreter, and others are available to the programmer. Even a programmer who does not directly use any of the parallel data structures will benefit from the system because *the language interpreter itself uses parallel data structure operations to run faster*. This approach ensures that programs can actually make use of the architecture's power and it greatly eases the burden of writing parallel programs.

In a conventional computer, the CPU can issue two "instructions" to the memory: *fetch* and *store*. Programmers must implement all their data structures and algorithms using just these two operations, along with the ability to do address arithmetic in the processor. APSA consists of a processor connected to a *heap memory* which executes about 30 data structure instructions in addition to *fetch* and *store*. Some of these heap memory instructions can do work that requires a loop if only *fetch* and *store* are available — *in a constant number of cycles*. Several examples of APSA data structure operations are given below.

Most approaches to multiprocessing view a computation as a given set of instructions that must be executed, and they try to partition the instructions into subsets that can be executed in parallel on separate processors. In contrast, APSA uses parallelism to reduce the number of instructions that must be executed, and it reduces the time complexity of several key algorithms by $O(n)$, where n measures the size of the problem.

APPLICATIVE LANGUAGES AND PARALLELISM

In most recent research on parallel computing, the underlying language is imperative (e.g., Fortran), and the underlying architecture comprises conventional processors, memories, and switches organized as a coarsely grained multiprocessor or vector processor. These assumptions limit the amount of parallelism that can be obtained because:

- Imperative languages place unnecessary constraints on the order and concurrency of primitive operations.
- Systems built from conventional processors and memories cannot fully exploit the fine grain parallelism that VLSI hardware makes feasible.

APSA produces parallelism through an applicative language running on an architecture with an extremely fine grained mixture of logic and storage. This serves several purposes, discussed below: (1) APSA makes the speed of an applicative program comparable to the speed of an imperative program by making the primitive applicative operations fast, (2) APSA can perform independent reductions in parallel, indicating that it may be able to exploit the high level parallelism inherent in applicative languages, and (3) the APSA hardware uses the ability of VLSI to mix logic and storage in regular layouts.

The main drawback of applicative languages is that they are usually slow, because conventional data structures based on *fetch* and *store* cannot support many primitive applicative operations efficiently. For example, a FORTRAN program can fetch the value of a variable simply by executing a *fetch* on a conventional machine, but fetching the value of a variable in an applicative language is much more complex and usually requires iteration. This phenomenon creates the illusion that the applicative language is inherently slower than FORTRAN, but the real problem is just a mismatch between the language and conventional architectures. APSA uses parallel data structure operations to make the primitives for an applicative language execute in a small constant time.

As it runs, an applicative program generates many expressions whose evaluation *cannot* affect the execution of other expressions. Several "parallel reduction machines" have been proposed to exploit this inherent parallelism. In many cases APSA is able to do parallel reductions, and further research in this area looks promising.

Since it performs data structure algorithms in the heap memory hardware, APSA combines substantial processing logic with every memory word. This organization takes advantage of the properties of VLSI circuits: a large heap memory consists of a simple regular pattern of memory word components. In addition, the heap memory could use wafer scale integration because of the simple interconnection of its components. A preliminary VLSI design of the heap memory is in progress.

PARALLEL DATA STRUCTURE ALGORITHMS

Many key algorithms for implementing applicative languages are generally considered to be inherently sequential. For example, indexing into a linked list, marking accessible words in a garbage collector, managing continuations and maintaining variable environments are all crucially important for applicative language implementation — yet these are implemented sequentially *even in proposed multiprocessors*. APSA implements all these operations quickly by providing special instructions in the heap memory that use hardware parallelism instead of software iteration.

The implementation of a combined list/vector data structure illustrates how APSA works. Vectors are normally implemented by storing adjacent vector elements in storage locations with consecutive addresses. This allows the address generation hardware in conventional memories to access an arbitrary vector element directly. However, it is very expensive to insert or delete an element in the middle of the vector, because that requires recopying much of the vector in order to make space for the insertion or to fill up the hole left by the deletion. The properties of linked lists are just the opposite: "indexing" into the middle of a linked list is slow, because it requires a loop that follows pointers from one element to the next, but insertion and deletion are fast because they require only a few pointer manipulations. Programmers know how to use vectors and linked lists effectively, but they also "know" that a data structure cannot combine the advantages of both.

The APSA heap memory can usually store a linear data structure compactly, similarly to the conventional vector representation. It can index any element directly, and it can also perform an arbitrary insertion or deletion in constant time by moving as many data elements as necessary out of the way in parallel. This provides the programmer with an extremely powerful data structure. Instead of thinking about how to schedule multiprocessors, the programmer thinks about how to use the flexibility of the data structure manipulation made possible by APSA. Lists and vectors are just special cases of the APSA data structure.

Parallel garbage collection shows another way that APSA speeds up algorithms, and it also illustrates how two levels of parallelism can be combined in a single algorithm. Garbage collection is one of the most crucial algorithms required to implement applicative languages, and its performance may determine whether an applicative programming system is usable. Other researchers have proposed several ways to construct parallel garbage collectors. Those techniques involve two processors running the interpreter and the garbage collector simultaneously, but the processor doing the garbage collection is strictly sequential. APSA also runs the interpreter and garbage collector in parallel. In addition to this high level parallelism, the APSA heap memory is able to mark many accessible words in parallel, achieving low level parallelism within the garbage collector. The resulting garbage collector is faster than it would be if either form of parallelism were used without the other.

APSA'S HARDWARE AUGMENTS CONVENTIONAL RAM ADDRESS DECODERS

The heap memory organizes all the memory cells into a linear shift register with data paths connecting neighboring cells. In addition, some of the memory instructions require a tree of combinational logic connected to the memory cells. Even with all this hardware, the APSA heap memory is similar in speed to conventional RAM memories, because RAM address decoding logic has the same time complexity: $\log N$ for N words.

This analysis explains the source of APSA's parallelism. Conventional memories get very little power from their address decoders: they just implement *fetch* and *store*. APSA uses its combinational logic tree to implement a variety of parallel data structure primitives.

EMULATION OF APSA USING FINE GRAIN PARALLEL MACHINES

In general, it is not clear how to implement LISP on parallel computers. However, the APSA research provides an approach for doing this. Instead of implementing LISP directly on the parallel computer, we can emulate the APSA heap memory, which is usually more straightforward. The processing elements of the real parallel computer can then execute the applicative data structure operations in parallel.

Following this method, APSA is currently being emulated using the Goodyear Massively Parallel Processor (MPP) [3, 6, 7]. The Connection Machine (CM) [1] would also be a good host for emulating APSA. This raises a number of interesting issues relating to the MPP and the CM. For example, the MPP processing elements can communicate only with their nearest neighbors, while the CM has long data paths. However, the MPP has an advantage that may compensate for its lack of connections between distant processors: the MPP communication algorithms are deterministic, synchronous and very fast. Although the CM can send messages directly over long distances, it cannot guarantee their arrival at a particular time. Thus the APSA emulation requires less overhead due to synchronization on the MPP than on the CM. We do not know yet what the relative performance of the MPP and the CM would be.

DISCUSSION

There is a tradeoff between how specialized a machine is and how successful it is on a wide range of applications. There are two ways of looking at APSA:

- It is a special purpose machine, and its intended application is the implementation of applicative language interpreters.
- It is a general purpose programming language host which can run algorithms for any application.

Thus APSA makes any applicative program run more efficiently. APSA's chief advantage is that it is easy for the programmer to obtain the benefits of its parallelism, because the programmer doesn't need to worry about breaking the problem into independent parts, or scheduling a number of processors, or reducing memory contention. However, there is a penalty for this ease of use: the fastest possible parallel algorithm for a particular problem may not be able to run on APSA. An important area for further research is investigating how to fit more specialized parallel algorithms into the APSA framework.

APSA achieves two levels of parallelism in its garbage collection algorithm: (1) simultaneous instructions issued by two processors and (2) parallel operations within the heap memory. Many other problems also exhibit several levels of inherent parallelism, and it will be fruitful to try to exploit as many levels as possible in one system. If it is possible to combine APSA's heap memory operations with multiprocessor parallelism, the resulting architecture might provide several levels of parallelism for a large set of problems.

REFERENCES

- [1] W. Daniel Hillis, *The Connection Machine*, The MIT Press, Cambridge, Mass., 1985.
- [2] John T. O'Donnell, *A Systolic Associative LISP Computer Architecture with Incremental Parallel Storage Management*, Technical Report 81-5, Computer Science Department, The University of Iowa, Iowa City, 1981.
- [3] John T. O'Donnell, "An Approach for Simulating an Applicative Programming Storage Architecture on the Massively Parallel Processor", Technical Report 179, Computer Science Dept., Indiana University, Bloomington, Sept. 1985.
- [4] John T. O'Donnell, "An Architecture that Efficiently Updates Associative Aggregates in Applicative Programming Languages", *1985 IFIP Symposium on Functional Programming Languages and Computer Architecture*, Springer-Verlag Lecture Notes in Computer Science 201, 1985.
- [5] John T. O'Donnell, "An Efficient Architecture for Implementing Sparse Array Variables," *Twenty-third Allerton Conference on Communication, Control and Computing*, Coordinated Science Laboratory, University of Illinois, pp. 986-995, October, 1985.
- [6] John T. O'Donnell, "Simulating VLSI Systems Using the Massively Parallel Processor", *Proceedings of the 1986 Summer Simulation Conference*, The Society for Computer Simulation.
- [7] Jerry L. Potter (ed.), *The Massively Parallel Processor*, The MIT Press, Cambridge, Mass., 1985.

Session 16: Recommendations

**Chairperson: Dharma P. Agrawal
 North Carolina State University**

DISCUSSION SESSION: "Future Directions in Computer Architecture and Software"

BY: Dharma P. Agrawal

CSNET ADDRESS: AGRAWAL@MCNC

COMPMAIL+ D.AGRAWAL

TOPIC: Instruction Set Considerations

- RISC vs. CISC
- Microprogramming
- Two level microprogramming
- Nanoprogramming

TOPIC: Custom Chips
Custom Chips vs. Off-the-shelf Component

- Lead time
- Impact of algorithm enhancement
- Changing algorithm
- Use for multiple applications

TOPIC: Memory Hierarchy

- Cache
- Second level cache
- Large shared memory and access time
- Multiport access conflict
- Differential growth in speed
(RAM fast, DISC slow)
- Associate mem.
- Hot spots
- Sync Primitives
- Interconnection Network

NO-A184 949

PROCEEDINGS OF THE WORKSHOP ON FUTURE DIRECTIONS IN
COMPUTER ARCHITECTURE. (U) BAYTELLE COLUMBUS LABS
RESEARCH TRIANGLE PARK NC D P AGRANAL ET AL. 30 AUG 86

S/S

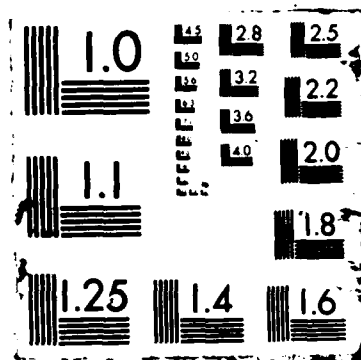
UNCLASSIFIED

ARO-86384-EL DAAG29-81-D-8188

F/G 12/5

NL





TOPIC:

Interconnection Networks

- Optical IN
- Cross bar switches
- Fault-Tolerant Reliability
 (very large network - 10,000 proc.
 for massively parallel M/C)
- Multistage interconnection network
- Multibus
- Serial/parallel
- Circuit switched/package switching
- Hypercube
- Tree
- CCC, Pyramid

TOPIC:

SIMD / MIMD / Reconfiguration Strategies

- use of switches and involved complexity
- WSI
- Data distribution
- Clock Skewing / data skewing
- Automatic Configuration
- Dynamically changing precedence graph

TOPIC:

Granularity Issues (based on instruction or data)

- compilation granulation
- execution granulation
- large grain
- small grain
- medium grain
- degree of parallelism
- computation communication trade-offs

TOPIC:

Granularity based on Processor Precision

- access to parallel machine

TOPIC:

Mapping Algorithm and Task Assignment

- matching algorithm & architecture
 - . choosing right algorithm
 - . given algorithm, choose architecture
 - . reconfigurable system for a given algorithm

- computation model
 - . correctness & accuracy
- effectiveness of measurement
- dynamic scheduling
- static scheduling
- performance parameters
- trade-offs
- benchmarking

TOPIC: Reusable and Retargetable Software

- Generic building block
- What do we do with code for older machine (uniprocessor)?
- How could they be modified to run on a newer parallel system?
- Automated tools?
 - . How & what?
- Retargetable code generator
- Software factories

TOPIC: Distributed Operating Systems

- Multiple kernels
 - .
- Coordination
 - . global resource management
 - specially heterogeneous
- Concurrency control
- Replication (some service) sharing
- Network V/S, Multicomputer O/S
- Global resource management
- Tradeoffs of kernel v/s high level mar.

TOPIC: Concurrency Control

- Database update
- Formal correctness
- Heterogeneous D.D. Base
- Synchronization
 - . Synchronous vs. Asynchronous Algorithms
- Performance Evaluation
- Non-serialized transactions
- Deadlock
- Shared Memory Access Hardware
 - . Fetch & Add: cost effectiveness and alternatives

TOPIC:

MIMD Parallelism and Support

- Shared Memory
 - . vs. local
 - . vs. hybrid
- Packet Switching
- Synchronization Overhead

TOPIC:

VLSI - Programming

- . How do you do it?
- . What needs to be done?
- Parallel Programming
- Experience with Parallel System & Languages
- Implicit vs. explicit parallelism & tradeoffs
- Machine dependent vs. independent

TOPIC:

Logic & Functional Programming

- For what applications?
- Logic Programming in distributed data bases
- AI
- Large-grain parallelism?
- Implementation
- Hardware support

Future Directions in Computer Architecture

**A Recommendation for the Army Research Office Workshop on
Future Directions in Computer Architecture,
April 1986**

E. Douglas Jensen

**COMPUTER SCIENCE DEPARTMENT
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PA 15213
412-268-2574**

1. Summary

This is a brief outline of the author's recommendation about one future direction which computer architecture ought to take. The focus is on lowering the preponderant evolvability and interoperability component of life cycle costs for real-time C³/battle management computer systems, through greatly improved architectural (particularly processor) modularity; an additional cost/performance contribution by such modularity is the enhanced feasibility of architectural algorithmic specialization (an important capability despite the misunderstandings of RISC proponents).

2. Recommendation

This recommendation is directed towards embedded computers for real-time supervisory control, particularly in DoD combat platform and battle management applications — it is applicable in some degree to many other contexts.

In real-time control systems, the computer requirements are ill-defined initially and continue to change across not just the design phase but even across the entire lifetime of the system; that lifetime is often decades in DoD and certain industry (e.g., electric utility) applications. The principle reasons for this are: that the application is complex and is not (perhaps cannot be) well understood; the application environment (e.g., the threat) varies according to geopolitics, doctrine, etc., requiring that the system functionality be altered, the performance improved, and the size increased; technology evolves, making some desirable system attributes possible and others more cost-effective, but also making older technology first exorbitantly expensive and then unavailable for further production or even maintenance, so internal and transparent substitution is called for; the computer is considered the ultimate recourse for accommodating system modifications, because "it's a simple matter of programming."

We will use the term *modularity* for the characteristic a computer system needs in order to be successful in these exceptionally demanding circumstances. Common synonyms include "interoperability", "evolvability", and "extensibility," all of which we prefer to consider system attributes which depend on modularity.

Both software (e.g., the use of standard high level languages, and increasingly, operating systems) and hardware play a part in computer system modularity. We are concerned here with hardware, and specifically with approaches for individual processors; these can then be utilized in various kinds of multiple-processor computers to achieve additional modularity.

The costs of insufficient computer system modularity are consistently estimated to be an order of magnitude larger than the initial design and procurement costs. But a high degree of modularity is almost never even one of the driving forces in processor architecture, which suggests that computer system designers generally exhibit a blind spot and misplaced priorities (e.g., the single-minded passion for throughput in millions of instructions per second), at least so far as the real-time control environment is concerned. Consequently, the technology for achieving modularity is very primitive, and much more research and experience are necessary.

One approach has been the family of upward-compatible computers: but even when there is effective compatibility among family members, the cost/performance and functionality alternatives are limited and have been chosen by the computer manufacturer (with his own constraints and objectives in mind), not by the users for their requirements.

More appropriate would be an expandable range of processor instruction set architectures, starting with a basic core of instructions similar to the RISC notion, except that neither single-cycle execution nor frequency of use are inherently the proper (much less sole) criteria for membership in this core. On such a foundation, further functionality is added as justified by cost-effectiveness tradeoffs to meet current system requirements — for example: performance (e.g., algorithmic specialization, not only for an application but also for the system, such as object invocation, interprocess communication, real-time scheduling); fault tolerance (e.g., support for atomic transactions, replication); software compatibility.

This suggests the use of microcode as a processor modularity technique, although historically it has been devoted primarily to instruction set interpretation, with fewer instances of vertical migration from the operating system level and fewer still from the application level. While the advantages of microcode have not yet been fully exploited (for various technical and non-technical reasons), as

currently envisioned it is inherently limited. The microinstruction set is bound to a particular collection of data path resources — the functional (e.g., arithmetic) elements, storage units (e.g., registers), and transfer paths — and often there are interpretation segments (e.g., macroinstruction fetch) which are hardwired. This permits only limited flexibility in creating instruction set architectures and functionally specialized microengines tailored to given requirements.

Malleability of the instruction set (exo-)architecture is only part of the solution, however; a high degree of modularity implies that its implementation (endo-architecture) must also be alterable — e.g., acceleration mechanisms such as pipeline stages and caches can be included when and where they are needed. At the present time, the instruction set architecture and its implementation are always bundled together in any given processor, whether or not it is microprogrammed.

Another form of modularity is found in the idea of co-processors, such as the popular floating point units for microprocessors. The interfaces to current incarnations of co-processors are improving, but they are still excessively constrained. Control is usually master/slave and highly stylized (e.g., no form of nesting or other composition), execution is synchronized to the instruction sequencing of the main processor, and there are too few shared resources between the processors. An 'extended function unit', which is ordinarily a simple synchronous adjunct to the processor's main arithmetic unit, can also be used for incorporating certain extensions.

Thus, the recommendation is that effort be directed toward seeking new exo- and endo-architectural concepts and techniques which overcome the modularity obstacles of conventional processors. It should be possible to have a physical and logical framework that remains constant, to which various special-purpose and general-purpose functional (data handling and control) hardware and microcode elements are added at system configuration time as cost/performance tradeoffs call for. This can be done in a manner which combines the best of both the RISC and CISC philosophies — complex functions implemented in hardware or microcode are faster than equivalent software, but do not slow down the execution of simple functions which require only a single cycle. It should be possible for functions to be migrated among software, microcode, and hardware levels transparently to the programmers, perhaps by eliminating the classical dichotomy between software invocation mechanisms (i.e., procedure calls) and hardware/microcode invocation mechanisms (i.e., op codes), making the invocation of every function identical regardless of its implementation.

FUTURE DIRECTIONS IN COMPUTER ARCHITECTURE AND SOFTWARE WORKSHOP

Comments by J. R. Burke, Chairman Session 5 Interconnection Strategies

Interconnection networks are required to provide communication links in a multiprocessor computer system. Whether such networks are static or dynamic, their throughput can become a bottleneck when the number of processors is large. The paper "Image Texture Classification with an Optical Crossbar Interconnected Processor," in the Interconnection Strategies session describes a development which may overcome this bottleneck. The crossbar is effected optically and dynamically using, for example, lenses or acousto-optic gratings to determine the optical path between an array of laser diodes (representing sources) and an array of photodiodes (representing destinations). Messages would be passed by modulation of the laser diodes.

The following recommendations were abstracted from the summary session where the workshop presentations were discussed:

1. The trade-off between design of custom chips vs. off-the-shelf ships will require modified algorithms.
2. Differential between processor speed and memory access time is increasing. On chip memory and overlapped pipelining of instruction and operands is helping. However, there are no revolutionary ideas as to how to fix this limiting bottleneck.
3. TI presented a paper on a fiber optic crossbar switch for a multiprocessor interconnection network. Because of large bandwidth of fiber optics, the bit rate can be much greater than on wire. Thus serial rather than parallel word transmission can be used. This significantly reduces the number of wires that need to be used. There was unanimous agreement that this technology should receive further support. As the DARPA contract with TI is about to end, I have suggested to John Zavada that he might want to consider a follow-on.
4. It was argued by some of the attendees that there are important applications where a multiprocessor system containing 10,000 processors would be useful (i.e., an imaging system using one processor per bit). In contrast to general belief, it was also argued that programs to exploit such a large degree of parallelism would be written if the machines were available.
5. As a corollary to #4, there is a need to provide multiprocessor machines to the community in order to accelerate experience in writing and testing programs to exploit parallelism for speed-up.
6. Automatic dynamic reconfiguration of processor interconnections to fit changing precedence graphs.
7. An evaluation of what are the desired performance measures for machines.

8. More appropriate benchmark algorithms/programs.
9. Theoretical measures for determining architecture effectiveness independent of software.
10. Tools to retarget software to new machines needed to avoid inefficiency of older languages such as Fortran on such machines. Billions of dollars has been invested in software development which is now OBE.
11. While distributed operating systems provide fault tolerance, efficiency and survivability, cooperation and correlation are problems. Techniques for global resource management need to be developed.
12. Tradeoffs between the uses of synchronization versus asynchronization need to be examined. Partial synchronization may be more important for some problems.
13. Need theory for non-serialized transactions.
14. Heterogeneous networks of machines are needed for some applications. Work on how to develop data bases for such networks is needed.
15. More work on Logic and Functional Programming.

Doyce Satterfield

SUMMARY

ADVANCED DISTRIBUTED ON-BOARD PROCESSING

The U.S. Army Strategic Defense Command has been over the past few years investigating advanced signal and data processing technology for obtaining target information with complex infrared (IR) sensors. The results of these investigations have culminated in the development of a very sophisticated hardware-in-the-loop signal and data processing testbed. This testbed facility consists of a Mosaic Sensor Emulation Unit (MSEU), an Auxilliary Sensor Signal Processor Unit (ASSPU), and an Advanced Distributed Onboard Processor (ADOP).

The ADOP system has a 15 MIP capability and comprises five processing elements or nodes which communicate on three, 1 M word/sec independent global buses. Each node contains three Central Processing Units (CPUs), three Floating Point Processors (FPPs), one million (1M) words of 17-bit memory, and a Global Bus Interface Controller (BIC).

The ADOP node architecture exercises four basic functions:

(1) A fixed point, MIL-STD-1750A instruction set architecture, CPU capable of executing the fixed-point Digital Avionics Instrumentation System (DAIS) instruction mix at 1 MIPS. The CPU implements a pipelined instruction queue and embedded memory management unit function to achieve this throughout, while addressing up to 1M words of memory. The basic CPU microcycle time is 200 nsec while the memory system access time is 350 nsec.

(2) A FPP serves as an adjunct to the CPU to facilitate achievement of 1 MIPS full DAIS floating point operation. Using the built-in function generator option of MIL-STD-1750A, special macro instructions can be coded to do special operations for signal processing operations. The FPP contains a hardware multiplier and an ALU which essentially "shadows" the CPU arithmetic Logic Unit.

(3) A four port local memory organized as 256K words by 17 bits (1 parity bit). A read-modify-write function facilitates semaphore operations required by the operating system. The system of node-local buses provides access to all of memory by each CPU.

(4) A BIC, which coupled with the real-time operating system, efficiently handles all global communications. The BIC accesses message queues in node-local memory and moves messages on the system of global buses. A logical addressing scheme allows the routing to be configured by the operating system without affecting application software.

The ADOP node structure allows virtually contention free processing of an application program when data sets/algorithms are mapped non-overlapping. To this end, a complete set of software tools has been developed, including a PASCAL compiler and a distributed operating system. The use of these tools provides a strict adherence to programming standards and configuration controls that make experimentation on the Multi-CPU, Multi-Node (distributed) architecture an achievable task.

SUMMARY

ADVANCED DISTRIBUTED ON-BOARD PROCESSING

(Continued)

The distribution of data bases, the mapping onto the architecture and the distributed operating system for this research and development program will be described at the 5 - 7 May 1986 workshop.

ARO workshop conclusions

Jon Mauney

I will restrict my comments to two software areas discussed at the workshop, leaving the other areas to those with more expertise.

Reusable Software

At the wrap-up session, two distinct topics were grouped under the heading of "Reusable Software." The first problem is the continued use of existing applications on new systems. This is an important problem because of the vast investment represented by the body of existing code. To protect this investment, the code must be adapted to new versions of a system, and to new operating systems, which may support different dialects of the programming language. Such transport, although important, does not seem to be a major research issue. The presentation by Takefuji and Dowell at the workshop described a fairly simple expert system for dialects of LISP; the Lexeme corporation is advertising a commercial product that translates among high-level languages such as Fortran, Pascal, and Ada.

A harder problem is efficient execution of existing code on very different hardware, such as vector and parallel machines. Again, this is important if the benefits of new architectures are to be extended to existing programs. This problem is addressed by much of the research into parallel compilers, especially the so-called "dusty-deck" compilers. There is still a lot to be done in this area. However, the general feeling of the languages-and-compilers researchers at the workshop was that the future lies with newer languages, more closely related to logic and functional programming than to Fortran.

The other Reusable Software problem is one of productivity in the writing of new applications. Programmers can be more productive if they can reuse parts of previous projects, or access a library of program building blocks, rather than having to start from scratch. This problem is the one addressed by the presentations in the workshop, and it is the kind of thing that the STARS program was set up to attack. This is clearly an important problem for the future.

Functional and Logic Programming

At the session on Logic and Functional Programming there was, of course, agreement that this style of programming is the direction to go in the future. Most people working on compilers and related software for parallel architectures have decided that functional languages have overriding advantages. It may be worthwhile studying ways to convince the everyday programmer that this is so.

The current research projects presented at this session were divided on the question of automatically detected parallelism versus explicitly programmed parallelism. One presenter began by stating that he rejected explicit parallelism and described how he avoided it, another described a parallel architecture for LISP using explicitly coded parallelism. This disagreement applies to conventional imperative languages as well as logic and functional languages. In the discussion

following, the consensus was that automatic detection of parallelism is preferable, especially if very large numbers of processors are available, but that there will always be applications that push the limits of performance and require close programmer control of parallelism. It would seem, therefore, that it will continue to be important to do research in automatic detection and exploitation of parallelism (both in conventional and in functional languages), and to determine what the best trade-off point is between automatic and explicit parallelism.

AT&T
Bell Laboratories

Crawfords Corner Road
Holmdel, New Jersey 07733
Phone (201) 949-3000

May 16, 1986

Professor Dharma P. Agrawal
Professor & Workshop Chairman
North Carolina State University
Department of Electrical & Computer Engineering
Box 7911
Raleigh, NC 27695-7911

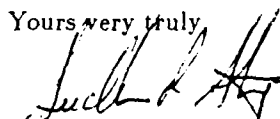
Dear Professor Agrawal:

I enjoyed attending the workshop on future directions in architecture and software. Although I spent only one day at the conference I found the papers very interesting. More importantly, the attendees represented the right mix of academia and industry and I found the discussions extremely fruitful.

The only area I would like to see covered in addition to the ones already discussed at the conference is the availability of new technologies. Development of new technologies such as wafer scale integration, optical computing, etc., can (and will), have a major impact on future computer systems.

Thank you very much for the opportunity to participate in the workshop. I look forward to interacting with you again in the future.

Yours very truly,



Sudhir R. Ahuja
Head
System Architectures
Research Department



DEPARTMENT OF THE NAVY

NAVAL TRAINING SYSTEMS CENTER

ORLANDO, FLORIDA 32816

2 Jun 86

Dr. Dharma P. Agrawal
Department of Electrical and Computer Engineering
North Carolina State University School of Engineering
Box 7911
Raleigh, NC 27695-7911

Dear Dr. Agrawal:

I must apologize for not answering your letter sooner relating to the recent ARO-sponsored workshop. Official travel seems to have occupied most of my time since returning from the workshop.

Speaking very candidly, I did receive benefits from participating; such as informal professional opinions and technical discussions in addition to the contacts made. However, as was discussed, the thrust of some of the presentations covered previous work rather than postulate future directions of the technology. Concerning my session (Session 4), I did feel that a portion of the presentations were within the workshop theme. The discussion by Dr. A. J. Smith of UC-Berkley concerning the future of memory hierarchies seemed to be of particular interest to the participants. The presentation of Dr. P. A. Subrahmanyane of AT&T Labs concerning parallel architectures for integrated system design tools was unique to a technology area (CAD/CAM) that is becoming increasingly important to the total computer technology arena.

For my own input to you, I am directly involved in exploratory development (6.2) and advanced development (6.3) projects related to special parallel processing architectures for unique very high speed processing. These also relate to future real-time artificial intelligence (expert systems) adjuncts to training systems. The potential of dataflow architectures to provide solutions to our long range requirements is also being addressed. It goes without saying that with the advent of the new standard DoD language, Ada, software developments should assume a new and different perspective for the future. Much work needs to be accomplished in this area.

My overall evaluation of the workshop was positive and I am of the opinion it served a need that should be exploited in the future. I would hope ARO would see the potential for and value of such workshops and assume the initiative in planning and executing them periodically. I would appreciate any opportunity to attend and participate in any such workshops.

I felt that you and your staff did a very fine job in planning and running the workshop. The location chosen was certainly unique and the facilities adequate. I will appreciate maintaining a contact with you in the future for the beneficial exchange of information related to the total area of computer technology.

Sincerely,

C. Forrest Summer

C. FORREST SUMMER, P.E.
Head, Advanced Computer Technology Branch
Research and Development Department

August 4, 1986

TO: Dharma P. Agrawal
Department of Electrical and Computer Engineering
232 Daniels Hall
North Carolina State University
Raleigh, NC 27695

FROM: R. R. Oldehoeft L-419
Lawrence Livermore National Laboratory
P. O. Box 808
Livermore, CA 94550

SUBJECT: Future directions in architecture and software

Thank you for your recent telephone call. I had intended a prompt response to your original request, but the end-of-semester rush combined with my relocation to California for my sabbatical leave prevented me from getting to it. I assumed my input was no longer timely, but I appreciate this opportunity nevertheless. Since I am not an architectural specialist, I can only comment from my point of view as an academic computer scientist and a software designer.

1. An important requirement for progress in parallel processing research is the provision of parallel processing testbeds for academic researchers. Since many will not have the capability or propensity for construction and maintenance of custom systems, this means that commercial systems are necessary. Fortunately, some are currently available, and they represent a variety of approaches to parallel processing. They include BBN's Butterfly, the Sequent Balance 21000, the Loral Dataflow system, Intel's Hypercube, the Ncube boards for use with an IBM AT microprocessor, vector processor add-ons, etc. More will surely appear.

Probably the best way to make these systems widely available (except for the cheapest) is to establish network access to "many" centers whose purposes are several. First, the centers must make availability and uptime of the systems and the network their prime concerns. Second, they need to help their clients in learning to use the systems. Third, they must individually and collectively disseminate useful information, tools and techniques to the entire community and the larger audience as well. Funding needs to be available for the establishment and continuing support of these centers, and the country's network capability needs to be expanded to reach more researchers.

There should be an educational aspect to these centers as well. We need many more people with an appreciation of parallel processing

2. On the software side, I'd like to discuss current issues with which I have some familiarity. The importance of applicative (functional) programming languages is beginning to be understood with the early results of the SISAL project, among others.
 - a) Additional language development is necessary to bring projects "up to speed" with respect to features that have been developed largely outside the domain of multiprocessing. Vector/array processing capabilities and modularity constructs for developing large systems are two examples.
 - b) The requirement that extant codes remain useful in the parallel functional world presents some interesting research opportunities. In particular, libraries of useful routines need to be accessible from applicatively written programs without sacrificing two advantages of writing in an applicative language. Parallelism must be reasonably well preserved, and the functional nature of computing must be protected from arbitrary interference by these imperatively written routines.
3. Finally, we need some studies in the granularity of parallelism different from the current approach. We use the granularity provided by a parallel processing system to obtain one point of data describing what works on a particular machine. Instead, we should find ways to easily alter the units of parallel work as well as the detailed hardware structures that do the work. This means that simulation studies will continue to be important for this research.

St. John

**Comments on the
ARO Workshop on Future Directions in
Computer Architecture and Software, May 1986**

L. N. Bhuyan

The Workshop organized by the Army Research Office was a great success. Although the scope of discussion was a little broad, there was a lot of personal rapport between the researchers working in definite areas. Particularly, the short presentations and discussions on the last day of the workshop were lively.

In general there was a feeling that the software was lagging behind the architecture. It is quite natural because not many parallel systems have been built for the researchers to do experiments with. It is my opinion that more effort needs to be spent on the design of operating systems, algorithms, and software both at basic research and experimental levels. At the same time, the ARO should continue supporting innovative ideas in architecture and encourage design and implementation of such architectures at the universities and industries.

COMMENTS ON COMPUTER A&S WORKSHOP: John Zavada

While a number of interesting talks were given at the workshop, only a few people seemed willing to express their ideas for new directions of research. Most of the talks were centered about an individual investigator's recent research activities. While the people at the workshop represented a cross section of the foremost researchers in this area, I think that the body of presented talks is an indication of the direction of current and near-term research and not directly a source of ideas for future directions. Of course, there were exceptions and I will comment on several that I noticed.

Col. Ward gave a very interesting talk concerning some of the DoD needs in the area of computer A&S. One of the items that impressed me dealt with the huge volume of software in DoD use and the effort to convert the data into a single language (ADA). While this doesn't sound very glamorous, I view this as a major problem confronting DoD operations. If existing codes for aircraft, missile defence, etc. are to be recoded accurately, either a great many computer people will have to be hired by DoD or a very long time will be required. This topic did not seem to attract much attention in the body of talks at the workshop. An area in which this need would seem to be critical is that of nuclear weapons. The software that is utilized for nuclear missile firings, guidance, and control must simply be of the highest quality and free of errors.

Another topic that Col. Ward mentioned was the need for machines capable of billions of operations per second. This topic is often mentioned in workshop meetings either on electronics or computers. I feel that there are a number of approaches to this problem but that nobody is quite sure on how to achieve this goal other than by brute force: smaller electronic devices, higher chip device density, networking of computers. It is much harder to develop new ways to process the data and to realize efficient parallel processing. Very few talks seemed to address new techniques for achieving such processing.

D. Jensen of CMU tried in his talk to simulate ideas concerning future research directions. I think that his comments questioning present computer techniques are valid. However, there wasn't much time for discussion and his talks often have a humorous aspect that may distract the audience from his central message. Still there should be more discussions on the present status of computer science and on the historical development of the area.

A. McAulay of TI gave an interesting talk on the use of optics for parallel processing and interconnects. While much of his talk was devoted to describing the hardware for achieving an optical crossbar switch, the actual use of such a device for computing was only briefly treated. I think that the use of optics in computer A&S is a largely undeveloped field. Clearly, optics is going to play a greater role in electronic devices and chips, but how is the architecture and software going to accommodate the added capabilities optics provide. Along with this use of optics is the question of reconfigurable architecture. Optics holds the promise of doing this if a number of material and manufacturing problems can be overcome. But only a small amount of work seems to be devoted to exploring how this reconfiguration can be utilized. G. Gray of VPI is doing work for fault tolerant operations but there must be other areas that could benefit from reconfigurability.

J. Staudhammer of Univ. of FL presented interesting results concerning 3-D color graphics. His talk centered on image generation and image transition. Symbolics is an important area of computer applications especially in the DoD arena. Many situations call for the manipulation of symbols and more

effort should be devoted to performing these operations efficiently. The implications on computer A&S should be studied.

In general, it is very difficult to predict the future, even for experts in a given area. Also, in an open forum many such experts may be reluctant to express their new ideas or feelings. Novel ideas are probably best reserved for funding agencies and Ph.D. topics. Consequently, it is easier to present results of studies that have been completed and submitted for publication.

An example of this can be found in the talk by S. Lundstrom of MCC.

While his talk was informative, it was a summary of work that has already been completed for the airline industry. Directions that MCC is following were not mentioned. Another aspects is that only a few people have the experience and knowledge to accurately judge trends and developments. Usually, these are the older people with years of work in the area. People like Jensen are a valuable source of information but there has to be suitable forum for them to expound their views and listen to opposing comments.

Perhaps there should be a follow-up workshop involving a smaller number of people drawn from the ones that were at Seabrook. Their objective would be to discuss the results of Seabrook I and to develop detailed scenarios as to the future directions of computer A&S.

John Zavada
ARO/ 18 Aug 86

ARO WORKSHOP CONCLUSIONS

Doyce Statterfield

May 20, 1986

Dear Dr. Agrawal:

I was very pleased to participate in the excellent ARO-sponsored workshop. You had it well organized and with many outstanding presentations. The Wednesday afternoon discussions were informative thanks to your taking the time to prepare the lead-in view graphs.

As far as feedback, I have given this considerable thought and from the ARO perspective of balancing their basic research program in computer science. My general feeling is that their program is well structured and balanced for their budget. I believe that we still have a lot of research to do before there will exist a fully distributed computing system. That is, a computing system that has distributed processors, distributed data base, and a distributed operating system, all required to realize the full potential of distributed computing.

One area that appears to be a bottleneck for high performance processing is that of memory access time. If there was one area that I would like to see more effort in, it would be this. Another important area is that of task assignment and mapping algorithms where the real world problems are a mix of sequential and parallel operations. I don't know that we are smart enough in this area.

My problem really is looking at basic research and not trying to make it immediately solve an existing real-world problem -- like a scan-to-scan algorithm for a passive sensor -- but this is my problem.

Again, thank you for considering me as a participant for your workshop. I look forward to seeing you again.

**COMMENTS ON ARO WORKSHOP ON FUTURE DIRECTIONS
IN ARCHITECTURE AND SOFTWARE, May 5-7, 1986**

C.A. Papachristou

- A. The general problem of mapping software algorithms or functions into parallel architectures has not been formulated adequately. Some speakers presented one or two interesting techniques but they were all limited to special purpose systems, e.g., image processing. More work needs to be done in this area.
- B. There were several talks on distributed computer systems but little was said about parallelism in embedded computer systems such as the ones incorporated in a large platform (e.g., an aircraft). There are differences between distributed and embedded systems with regard to capabilities for parallel processing. Embedded microsystems require parallelism at the microlevel, i.e., configurable microarchitectures that are amenable to VLSI and sophisticated microcontrol designs to support these structures. A research program in these areas is needed.
- C. There is a need of formal synthesis techniques and tools for multiprocessing architectures. There are several tools with the capability to automate or semiautomate the synthesis of processors; there are no such techniques for multiprocessors. At present, multiprocessors are put together basically by intuitive and empirical methods and then the design is evaluated by analysis or simulation. With the expected proliferation of parallel architectures, there will be a need to formalize the synthesis of such machines on the basis of formal requirements, specifications and design processes.

Following are ten areas which I feel should be supported by the ARO research programs in Computer Science over the next five years or so. Several of these areas were discussed during the team leaders meeting (I missed it but I have a copy of the transparencies). No particular order of significance is assumed.

1. Instruction set architectures and microprogramming
2. VLSI architectures and optical technologies
3. Embedded systems - parallelism at the microlevel
4. Distributed systems (including Operating systems)
5. Concurrency and concurrent systems (both synchronized and asynchronous)
6. Mapping algorithms into multiprocessor architectures
7. Interconnection networks and parallel architectures
8. Synthesis tools for multiprocessor and architectures
9. Reusable software design
10. Programming issues (including parallel and logic programming)

Comments on

Future Directions in Computer Architecture and Software

I have made my report in the form of the following comments:

1. The remark was made that "software is behind hardware". To correct this situation it is necessary to encourage organizations to buy/build hardware and experiment on it, i.e., work with real systems and learn. That is, the problem is to some extent managerial. Only when large numbers of organizations have such machines and do work on them will the appropriate directions be learned.

Let's get on with it and build the machines; then the software will catch up. Software certainly won't catch up without the practical experience gained from working with actual machines.

2. Another area involves the time and expense of producing large parallel systems. Such systems require custom chips which are as expensive and time-consuming to make as software. Combined with the software costs, the time from concept to system availability is far too long and far too expensive to do a great deal of experimentation.

Some kind of design tools are needed to translate ideas into reality in a much more automatic fashion than is currently possible.

3. A topic not mentioned at the workshop is packaging. The impact of packaging is much more than is commonly recognized. For example, the comment was made that "memory speeds are falling behind that of the cpu". The reason is not that memory is not fast enough, since memory is currently available that is even faster, in proportion to cpu speeds, than 10 years ago. For example, 25 ns memory is currently available in 64K bit chips.

The problem is packaging. By the time the memory is mounted on boards in sufficient quantity, the effective speed of the above 25 ns chips is almost down to 100 ns (in round numbers). This can be improved upon only by packaging. When such 25 ns memory can be used effectively, then the problems for multiprocessors brought about by caches (for example, the cache consistency problem) are solved -- simply eliminate caches (one can even pipe the memory and the processors for even greater resource utilization).

4. The problem of "RISC verses CISC", and the problem of "instruction set architecture" are not problems at all. I agree with the comment that "it is a problem of good engineering" and nothing more.

Research directions of interest to ARD:

The key aspects of computer development in the next decade, and for most Army applications, especially involving widely distributed resources and personnel, will be.

- 1) The explosion in COMMERCIALLY available parallel electronic processors;
- 2) The rapid development of new algorithms, efficient on these widely available parallel machines;
- 3) Improvements in parallel programming techniques and languages;
- 4) Computer Aided Design tools that will aid in rapid application of new computer technology, such as VLSI custom integrated circuitry;
- 5) Computing power enough to finally realize much better human interfaces especially high resolution color graphics, realtime speech generation and understanding, and tactile/motion-detecting sensors;
- 6) The increasing importance of communication and switching based on fiber optics that will push to become very fast, full-fledged optical processing within 10 years.

In summary, the main two ways to get better computing will be through parallelism and though much faster individual processor circuitry.

The short-term parallelism limits are about 1,000 powerful (1 MIPS +) processors or about 100,000 1-bit processors for aggregates of 1 to 10 billion operations per second (1 -10 BIPS). Within 10 years, the fastest parallel computers will provide 100 to 1000 BIPS from about 100,000 heavy processors in massive systems sharing many levels of memory or about 10,000,000 simple processors - essentially very simple eye/brain combinations. The new algorithm, programming, and language issues will be the key to harnessing parallelism, although some work will be needed in improving hardware switching and control mechanism to allow rapid access to many level of shared memory in massively parallel systems. Creating faster individual processors will be a matter of first extending silicon VLSI technology to gallium arsenide for the first 10 fold speed up, then slowing developing optical switching, memory, & processing for a massive 1,000 to 100,000 fold improvement in speeds. This will be mainly very basic hardware research in the next few years. Computation speedups with optical computing will come both from faster signalling and much easier massive parallelism with optical distribution on data, but speeds in 10 years probably will be in the 10 to 100 BIPS range only. Programming and use issues will come more slowly.

I hope this helps. Let me know if you need more evaluations.

Larry Wittie, 19 Aug 1986

Dear Dharma,

Thank you for your letter of May 9 and the list of topics on which we should comment. I am sorry for not responding earlier. (I have been in and out of town in May.) Following are my general comments. Please let me know if you want me to elaborate on what is said here or to provide more references. Do you want the copy of the transparencies used in the discussion session returned with some kind of markings

Regards,

Jane

Although most of the position papers delivered in the Workshop on Future Directions in Software and Architecture are concerned with hardware aspects of computer architecture, the attendees whom I had opportunity to talk with informally during the Workshop agreed that the most pressing problems in the future are in software design and development. How to write programs for parallel machines and distributed systems, how to maximize concurrency, how to increase reliability of the overall systems, and how to make software reusable are examples of these pressing problems. But among all these problems, the most important one of them, in my opinion, is how to bring together hardware design and software design and engineering disciplines in order to find good overall system design, analysis and synthesis methods.

Currently, computer hardware designers and software designers almost always work in insolation from each other, rather than taking a system-design-oriented approach. The majority of the papers given at the Workshop demonstrate this fact quite well. This traditional approach of partitioning the system design problem into design of machine instruction sets, memory hierarchy, interconnection networks, control structures, scheduling and resource management algorithms, etc. has served us well thus far. It was necessary to understand the design issues concerning individual components of any system before one can address the design issues concerning the overall system. Many of the well adopted system design approaches, (e.g., the layered approach to design of distributed systems and computer networks,) encourages the partitioning of systems into components and attacking the design and development of components separately.

In recent years, as computer systems become more complex and their applications more critical, higher performance, fault tolerance and availability become more essential. Addressing the performance and reliability of subsystems independently of each other often results in lack of overall efficiency and robustness. It is time to develop top down, integrated approaches to system (software and hardware together) design. My paper on distributed, macro dataflow architecture is an attempt in this direction, in general. The view of the distributed system as consisting of a collection of servers which can be invoked to perform different computation and communication primitives discussed in my paper is consistent with the view pointed presented the in the paper entitled "An approach to decentralized computer system" by J. M. Gray, IEEE Transactions on Software Engineering, June, 1986.

Dharma.

Below are some of my general comments about recommendations to the Army (ARO, and others) about funding priorities. This is not particularly an ordered list, and obviously reflects my research interests.

By the way, thanks for your hard work on the workshop. It generally went well, despite the housing problems. A future workshop, however, should probably attempt to select a higher quality set of attendees. Maybe a more personal and strongly solicitation would have been appropriate?

thanks again for your efforts.

Alan Smith

Memory Hierarchies:

Extremely important topic. Highest funding priorities to CPU Cache Memories and Disk Cache. The former is important for high performance CPUs, as are used in many weapons, guidance systems, and C&C systems. The latter is important to large data processing installations such as the Army has at its larger bases. Also important is optimization of memory systems in distributed systems. Memory interconnection networks are worth some study, especially in conjunction with (a) cache memories and (b) implementation. More pure modelling studies are probably not called for at this time.

Instruction Set Considerations

My impression is that the military computer architecture standard, which I've actually never looked at, is 16-bit. If so, it is time for a 32-bit standard.

Further research on RISC vs. CISC is called for, with more comparisons and sound scientific studies.

The increased use of custom chips as opposed to large software systems and standard processors should be studied.

There is the need (as we discussed at the end of the workshop) for the Army to have a Computing Research Advisory Board, to be used to oversee the award of research grants and contracts, through ARO and other organizations (E.g. BMD Huntsville). I can suggest appropriate people (including myself) if requested.

Work on reconfiguration strategies is unlikely to be worth funding until someone has a working, usable, programmable reconfigurable computer. At that point, some experiments and further studies are called for.

The Army (or DOD) should allocate a lot of money (>\$10,000,000) to place parallel machines (e.g. Encore, Sequent) in universities and research labs, for studies of parallel software development, granularity issues, etc.

My personal opinion is that small grain parallelism is a poor idea for most applications and won't work. Likewise, I think that multiprocessors with large numbers of processors (currently, more than 100) are a very questionable undertaking until we learn how to make and use MPs with small numbers of processors. Such systems are likely to be of use only for certain specialized applications.

Mapping algorithms and task assignment - this needs to be studied in the context of real hardware on which to experiment. (see above). More abstract studies should be deemphasized for the time being.

Reusable and retargetable software is an idea whose time never seems to come. Everyone wants to write software for OTHER people to reuse. No one ever wants to use existing software. I would not put money into this.

Conversely, I would put money into experiments on software productivity where existing personnel are given STATE OF THE ART software environments. (e.g. SUN workstations, good ADA compilers, syntax directed editors, interactive debuggers, large mainframe), and are to work on regular, normal, military software projects. My suspicion is that the software environments are probably lousy, and that the Army is not using existing and known techniques.

The development and optimization of distributed operating systems is important. One important project would be to fund development of distributed (generally available) UNIX. (E.g. Berkeley 3.0 BSD). What is needed is a real distributed operating system on which people can experiment. (see also above about making parallel systems available). This is a generally important area.

Currency control - currently too many theoretical studies, too little actual data, implementation, and/or practical studies.

Comments on

Future Directions in Computer Architecture and Software

Mary Jane Irwin

This write-up discusses research directions in only four of the session topic areas.

Two (custom chips and granularity issues)

are directly related to our ARO supported research effort at Penn State and two are areas which are of interest to the researchers at SRC.

Custom Chips

Research has evolved in the past several years from a single chip design to systems design, an assemblage of many custom chips as well as many off-the-shelf components.

In systems design, not only must the individual custom chips work as required but they must also be configured into a system which requires clock and data distribution across board(s).

As chips get faster and systems more complex, both clock skew and data skew may become problems.

Thus, researchers have to learn to think and look beyond the single chip.

To do this requires more expensive design equipment and supporting personnel.

CAD tools and equipment which support systems design are largely missing at Universities, as such tools are very expensive and not generally available for free to Universities.

At least in Computer Science Department's, supporting personnel are also not usually available.

Special purpose systems based on unusual number representations (residue, logarithmic, and signed-digit) are now in production.

Memory Hierarchies and Interconnection Networks

Research in the areas of memory hierarchies and interconnection networks has been directed, in part, at reducing memory latency.

(Interconnection networks when used to connect processors and memories in an MIMD system; caches in memory hierarchies.)

Both of these approaches have inherent limits.

Data consistency problems can arise when using caches and local memories.

As the number of processors and memories increase, interconnection networks which are more scalable (e.g. rooted networks like

H-trees and log n networks like ccc's) are more promising than those that don't scale at all (e.g., buses and rings).

Network hotspots, combining networks (like Fetch-and-ADD), circuit versus packet switching, layout in VLSI, and fault tolerance of interconnection networks are also important research issues.

Avoiding the memory latency problem with new CPU architectural techniques is even more promising.

By issuing the next instruction to the instruction pipeline from a different process every cycle, the memory dependency problem can be reduced significantly or removed altogether.

This technique requires enough registers to allow free (or almost free) context switching every cycle; each process must own its own register set when "active."

This technique is used in the HEP machines to reduce the impact of memory latency.

How many processes need to be active, how are the register sets managed, what happens when the data has not yet arrived and the next instruction in the process is due to be issued are research issues.

Grainularity Issues

Grainularity can relate to the physical size of the processors, the process size, or the context switch cost.

Processors which are fine grain are attractive because of the obvious VLSI advantages.

They probably have bit or digit serial data transmission and can benefit greatly in speed if the operations can also be pipelined at the bit or digit level.

Processes which are fine grained ("light weight") can more easily be created, destroyed, and exported in an MIMD environment and may have advantages over the more traditional "heavy weight" processes. More research needs to be done in this area.

Fine grained context switching allows one to avoid the memory latency problem with round robin instruction issuing as discussed in the section above.

Issues in MIMD Architectures.

Both message passing systems (like the iPSC and NCUBE) and shared memory systems (like the BBN Butterfly and RP3) are now in production.

The debate continues as to which is the better approach for an MIMD machine.

However, building the machine is only one part of the problem.

The design of parallel languages to accommodate such architectures and the development of parallel algorithms, both very important research areas,

would be encouraged by making such machines available to the University researcher.

Is a hybrid system feasible, one which relies on message passing for those operations which work best in that environment and one which relies on shared memory for the others.

ARO Directions Workshop

H. J. Siegel 8/22/86

List of topics I feel are important to pursue INCLUDES (in random order):

RISC vs CISC architecture - when is each appropriate; what are the tradeoffs.

parallel processing system memories - local mem. vs global shared mem. vs hybrid approaches; "hot spots" in shared memory accessing; use for synchronization primitives; when is the "fetch and add" primitive cost-effective?

interconnection networks for large-scale parallel processing - use of optics for implementation; fault tolerance; tradeoffs among packet-switching, circuit-switching and hybrid approaches; tradeoffs between multistage cube and hyper cube approaches.

SIMD/MIMD/reconfigurable systems - operating system methods for switching between SIMD and MIMD modes, and for performing other reconfigurations; language features to specify reconfiguration; hardware support for efficient reconfiguration; mapping algorithms to reconfigurable systems.

mapping algorithms and architectures - models of algorithms and architectures and how they interact; given parallel architecture - which algorithm approach best; given algorithm - which architecture best; given a reconfigurable system - which configuration/algorithm pair best.

parallel programming - what language features needed for efficient "explicit" specification of parallelism; what features needed for effectively compilable "implicit" specification of parallelism; portable parallel languages for sharing work; common methods for expressing parallel algorithms so that researchers can share and communicate results among themselves more easily; tradeoffs between the efficiency of writing machine dependent "explicit" specification of parallelism programs vs machine independent "implicit" specification of parallelism; developing and documenting a set of parallel programming techniques.

LIST OF ATTENDEES

1	Aggarwal, Balraj 919-543-5221	IBM Coporation RTP, NC 27709
2	Aggarwal, Prem 919-549-7226	Northern Telecomm. RTP, NC 27709
3	Agrawal, Dharma P. 919-737-2336	Dept. of Electricl and Computer Engineering North Carolina State University Raleigh, NC 27607
4	Allen, Kelth R. 803-656-3444	Dept. of Computer Science Clemson Univ. Clemson, SC 29634-1906
5	Alonso, Rafael 609-452-3869	Dept. of Computer Science B224 Engr. Quadrangle Princeton Univ. Princeton, NJ 08544
6	Andrews, D. 415-941-3912	Advanced Decision Systems 201 San Antonio Circle Suite 286 Mountain View, CA 94040-1289
7	Barbacci, Mario 412-268-7704	Software Engr. Institute CMU Pittsburgh, PA 15213
8	Bhuyan, L.N. 318-231-6284	The Center for Advance Computer Studies University of SW Louisiana P.O. Box 44330 Lafayette, LA 70504
9	Burke, J. Richard 919-549-0641	P.O. Box 12211 Research Triangle Park, NC 27709-2211
10	Chen, Pin-Yee 609-866-6531	RCA Advanced Tech. Lab. Moorestown, NJ 08057
11	Cheng, Raymond 609-722-2799	RCA Corporation Marne Highway Moorestown, NJ 08057
12	Chung, Moon Jung 518-266-8326	Dept. of Computer Science and Center for Integrated Electronics RPI Troy, NY 12180-3590
13	Cook, Robert P. 804-924-7605	Dept. of Computer Science Univ. of Virginia Thornton Hall Charlottesville, VA 22903
14	Dasgupta, Partha 404-894-2572	School of Info. & Computer Science Geogia Institute of Technology Atlanta, GA 30332
406		

15	de Maine, P.A.D. 205-826-4330	Computer Science and Engr Dept. Auburn Univ. Auburn, AL 36849
16	Desai, B.C. 514-848-3025	Computer Science Dept. Concordia Univ. Montreal, Quebec H4B 1R6 Canada
17	Dowell, Mike 803-777-5099	Dept. of Electrical and Computer Engr. Univ. of South Carolina Columbia, SC 29208
18	Dyment, Doug 519-888-4451	Dept. of Computer Science University of Waterloo Waterloo, Ontario, Canada N2L 3G1
19	Efe, Kemal 314-882-4480	218 Math Science Bldg. Univ. of MO-Columbia Columbia, MO 65211
20	Elderhorst, Linda 301-863-6565	BDM Corp. 20 Coral Drive Lexington Park, MD 20653
21	Elmagarmid, Ahmed 814-863-1047	Dept. of Electrical Engineering Penn. State University University Park, PA 16802
22	Fendrich, John W. 309-676-7611	Dept. of Computer Science Bradley Univ. Peoria, IL 61625
23	Gao, Guang R. 617-253-6078	NE43-253 Lab for Computer Science MIT Cambridge, MA 02139
24	Gehringer, Edward 919-737-2336	ECE dept. NCSU Raleigh, NC 27607
25	Gilmer, John B., Jr. 703-247-3300	The BDM Corp. 7915 Jones Branch Dr McLean, VA 22102
26	Gostelow, Kim 518-387-5805	GE Research and Development Center KWC-286, P.O. Box 8 Schenectady, NY 12301
27	Gray, F. Gall 703-961-7059	Dept. of Elec. Engr Virginia Polytechnic Institut and State Univ. Blacksburg, VA 24601
28	Green, C. Ronald 919-549-0641	P.O. Box 12211 Research Triangle Park, NC 27709-2211

29	Hand, Steve 919-737-2336	Dept. of ECE NCSU Raleigh, NC 27695
30	Helal, A. 814-863-1047	Dept. of Electrical Engr. Penn State Univ. University Park, PA 16802
31	Hwang, phil 202-696-4312	CNR 800 Quincy Street Arlington, VA 22217
32	Irwin, Mary Jane 301-731-4145	Institute for Defence Analysis 4380 Forbes Blvd. Lanham, MD 20706
33	Jamieson, Leah H. 317-494-3653	School of Elec. Engr. Purdue University West Lafayette, IN 47907
34	Janakiram, J. 919-737-2336	Dept. of ECE NCSU Raleigh, NC 27695
35	Jensen, E. Douglas 412-268-2574	Dept. of Elec. and Computer Engr. Carnegie-Mellon Univ. Pittsburgh, PA 15213
36	Johnson, Ralph E. 217-333-0135	Dept. of Computer Science 1304 West Springfield Ave. Urbana, IL 61801
37	Joy, Edward J. <div></div>	2 Mason Street Troy, NY 12180
38	Kaplan, Ian 619-560-5888	Loral Data Flow Group Loral Instrumentation 8401 Aero Dr. San Diego, CA 92123
39	Kaplan, Simon M. 217-244-0392	Dept. of Computer Science 1304 West Springfield Ave. Urbana, IL 61801
40	Kozaczynski, W. 312-996-2676	1120 Science and Energy Offices University of Illinois Chicago, IL 60612
41	Kumar, Vipin 512-471-4353	AI Lab Computer Science Dept. Univ. of Texas at Austin Austin, TX 78712
42	Leu, Ja-Song 919-737-2336	Dept. of ECE NCSU Raleigh, NC 27695

43	Lin, Kwei-Jay 217-333-4428	Dept. of Computer Science University of IL at Urbana-Champaign Urbana, IL 61801
44	Lindstrom, Gary 801-581-5586	Dept. of Computer Science Univ. of Utah Salt Lake City, UT 84112
45	Liu, Jane W.S. 217-333-0135	Dept. of Computer Science 1304 West Springfield Ave Urbana, IL 61801
46	Lundstrom, Steve 512-834-3320	Vice President, MCC 9430 Research Blvd. Echelon Building #1, Suite 200 Austin, TX 78759
47	Mahgoub, I. O. 919-737-2336	Dept. of ECE NCSU Raleigh, NC 27695
48	Manwaring, Mark [REDACTED]	ECE Dept. Washington State University Pullman, WA 99164-2210
49	Mauney, Jon 919-737-7889	CSC NCSU Raleigh, NC 27607
50	McAulay, Alastair 214-995-0345	TI Inc. Computer Science Center P.O. Box 226015, MS 238 Dallas, TX 75266
51	McDonald, John F. 518-270-6033	Center for Integrated Electronics RPI Troy, NY 12181
52	Meador, Jack L. 509-335-8067	ECE Dept. Wash. State Univ. Pullman, WA 99164-2210
53	Mehrotra, Ravi 919-737-2336	ECE dept. NCSU Raleigh, NC 27607
54	Michalson, William [REDACTED]	Raytheon Corp. Equipment Division Boston Post Road Sunbury, MA 01776
55	Mielke, Roland R. 804-440-3741	Dept. of Elec. Engr ODU Norfolk, VA 23508
56	Mudge, Trevor N. 313-764-0203	Dept. of EE and CS Univ. of Michigan Ann Arbor, MI 48109

57	NI, Lionel M. 517-353-4386	Dept. of Computer Science Michigan State Univ. East Lansing, MI 48824
58	Nickel, Vincent V. 213-217-3710	14 Pinto Lane Rolling Hills Estates, CA 90274
59	O'Donnell, John T. 812-335-0739	Computer Science Dept. Indiana Univ. 101 Lindley Hall Bloomington, IN 47405-4101
60	Oldehoeft, R.R. 303-491-7017	CS dept. Colorado State Univ. Fort Collins, CO 80523
61	Omar, S.I. 613-564-5449	Dept. of CS Univ. of Ottawa Ottawa, Canada K1N 9B4
62	Owens, Robert M. 814-863-0392	Dept. of Computer Science Penn State Univ. University Park, PA 16802
63	Page, Edward W. 803-656-2398	Dept. of Computer Science Clemson Univ. Clemson, SC 29634-1906
64	Papachristou, C.A. 216-368-5277	Computer Engr. and Science Case Western Reserve Univ. Cleveland, OH 44106
65	Pargas, Roy P. 803-656-3444	Dept. of Computer Science Clemson Univ. Clemson, SC 29631
66	Prommel, Joan 505-667-6961	Los Alamos National Lab C-10 MS B296 P.O. Box 1663 Los Alamos, NM 87545
67	Przybylinski, S. M. [REDACTED]	General Dynamics Data Systems Division P.O. Box 85808 V2-5530 San Diego, CA 92138
68	Raney, Steven D. 303-977-4143	Martin Marietta Aerospace MS 0427 P.O. Box 179 Denver, CO 80201
69	Satterfield, Doyce 205-895-4431	Dept. of the Army, Office of the Chief of Staff U.S Army Strategic Defense Command - Huntsville P.O.BOX 1500 Huntsville, AL 35807 - 3801
70	Shin, Kang G. 313-763-0391	Div. of Computer Science and Engr. Dept of EE and CS The University of Michigan Ann Arbor, MI 48109

71	Shirazi, Behrooz 214-692-2874	Dept. of CS School of Engr. and Applied Sc. Southern Methodist Univ. Dallas, TX 75275
72	Siegel, H.J. 317-494-3444	School of Electrical Engineering Purdue University West Lafayette, IN 47907
73	Singhal, Mukesh 614-422-4635	Dept. of Elect. Engineering The Ohio State Univ. 2036 Neil Avenue Mall Columbus, OH 43210
74	Smith, Alan Jay 415-642-5290	Computer Science Div. EECS dept Univ. of California Berkeley, CA 94720
75	Sridharan, N.S. 617-497-3366	BBN Labs 10 Moulton St. Cambridge, MA 01742
76	Staudhammer, John 904-392-4910	College of Engr. Univ. of Florida Gainesville, FL 32611
77	Stoughton, John W. 804-440-3741	Dept. of Elec. Engr. Old Dominion Univ. Norfolk, VA 23508
78	Strip, David R. 505-844-3962	ORG 6228, Sandia National Lab. P.O. Box 5800 Albuquerque, NM 87185
79	Subrahmanyam, P.A. 201-949-5812	AT&T Bell Lab. Crawfords Corner Rd. Holmdel, NJ 07733
80	Sugla, Binay 201-949-0850	4G604 AT&T Bell Lab. Crawford Corner Rd. Holmdel, NJ 07733-1988
81	Summer, C. Forrest 305-646-4437	Head, Advanced Computer Technology Branch Code 741 Naval Training System Center Orlando, FL 32813
82	Szymanski, Bolek 518-266-8326	Dept. of Computer Science RPI Troy, NY 12180
83	Taylor, Fred J. 904-392-0911	Dept. of Elec. Engineering Larsen Hall Univ. of Florida Gainesville, FL 32611
84	Testard-Vaillant, F. 433-625-25	Laboratoire d'Informatique Theorique et Programmation 5 Place Jussieu 75252 Paris Cedex 05, France

85	Tracz, William J. 415-723-1089	Computer Science Lab Stanford Univ. Stanford, CA 94305
86	Waksman, Abraham 215-787-1911	Temple Univ. Philadelphia, PA 19122
87	Wallace, C.S. 613-541-3900	Dept. of Computer Science Monash Univ. Clayton, Victoria Australia 3168
88	Ward, Frank [REDACTED]	Office of Undersecretary OUSD (R&AT) Room 3-E114 Pentagon Washington, DC 20301
89	Wise, David S. 812-335-4866	Computer Science Dept. Indiana Univ. Bloomington, IN 47405-4101
90	Wittle, Larry D. 516-246-8215	Dept. of Computer Science SUNY at Stony Brook Long Island, NY 11794-4400
91	Yu, Clement 312-996-2318	Dept. of Elec. Engr and Computer Science Univ. of Illinois at Chicago Chicago, IL 60680
92	Zavada, John M. 919-549-0641	U.S Army Research Office P.O. Box 12211 RTP, NC 27709-2211
93	Zee, Benjamin 312-929-7903	IW IA-232 AT&T Information Systems 1100 East Warrenville Rd. Naperville, IL 60566
94	Zwaenepoel, Willy 713-527-8101	Dept. of Computer Science Rice Univ. Houston, TX 77001

<i>Author Index</i>	<i>Section</i>	<i>Page No.</i>
Ahuja, Sudhir R.	5.1	86
Allan, S.J.	7.2	139
Allen, R.R.	14B.5	301
Alonso, Rafael	10.2	202
Andrews, D.	15A.1	332
Barbacci, Mario	13.3	261
Bhuyan, L.N.	5.3	99
Cann, D.C.	7.2	139
Cheng, Raymond	2.3	29
Chung, Moon Jung	14A.3	276
Cook, Robert P.	2.2	21
Dasgupta, Partha	6.3	122
de Maine, P.A.D.	15B.3	346
Delp, E.J.	8.1	147
Desai, B.C.	6.2	114
Doulan, B.	15B.2	343
Dowell, M.	14C.1	316
Efe, Kemal	8.3	163
Elmagarmid, Ahmed	12.2	228
Fendrich, John W.	14B.4	299
Friedman, R.	15C.4	370
Gao, Guang R.	15C.3	367
Gehringer, Edward	14A.4	279
	14A.1	270
Gendreau, T.B.	14B.1	290
Gilmer, John B., Jr.	3.3	55
Gray, F. Gall	6.1	103
Green, P.E.	14C.5	328
Greub, H.	15B.2	343
Grimshaw, A.	8.2	155
Guh, K.C.	12.1	220
Heial, A.	12.2	228
Hurson, A.R.	12.2	228
Husson, A.R.	15B.4	349
Jamieson, Leah H.	8.1	147
Jensen, E. Douglas	2.1	17
Jiang, O.	15B.3	346
Jodis, S.	15B.3	346
Johnson, R.E.	9.2	178
Juels, Ronald J.	14C.5	328
Kabat, W.C.	15C.4	370
Kaplan, Ian	7.1	131
Kaplan, Simon M.	9.2	178
Kozaczynski, W.	15C.4	370
Kumar, Vipin	14C.4	326
Leblanc, R.J., Jr.	6.3	122
Leong, S.	15B.3	346
Lin, Kwei-Jay	15A.3	336
Lin, Yow-Jian	14C.4	326
Lindstrom, Gary	15C.1	361
Liu, Jane W.S.	8.2	155

<i>Author Index</i>	<i>Section</i>	<i>Page No.</i>
Lodo, A.A.	14A.3	276
Lundstrom, Stephen	11.2	214
Manwaring, Mark	15C.2	364
Marshall, D.A.	13.2	253
Masapati, G.H.	15B.5	353
Mauney, Jon	15A.2	334
McAulay, Alastair	5.2	91
McDonald, John F.	15B.2	343
Meador, Jack L.	15C.2	364
Mehrotra, Ravi	14A.4	279
Merchant, H.	15B.2	343
Mielke, Roland R.	14B.7	304
Mudge, Trevor N.	14A.2	278
Ni, Lionel M.	14B.1	290
O'Donnell, John T.	15C.5	372
Oldehoeft, R.R.	7.2	139
Omar, S.I.	15B.5	353
Owens, Robert M.	3.1	38
Page, Edward W.	14A.5	284
Papachristou, C.A.	15B.1	340
Pargas, Roy P.	14B.5	301
Pose, R.	14B.2	293
Przybylinski, S. M.	9.3	186
Raney, Steven D.	13.2	253
Satterfield, Doyce	4.2	72
Schreiber, R.	15B.2	343
Shin, Kang G.	14A.6	287
Shirazi, Behrooz	15B.4	349
Siegel, H.J.	8.1	147
Singhal, Mukesh	12.3	236
Smith, Alan Jay	4.1	62
Sridharan, N.S.	14C.2	319
Staudhammer, John	14B.3	296
Stoughton, John W.	14B.7	304
Styles, R.C.	4.2	72
Subrahmanyam, P.A.	4.3	78
Sugla, Binay	5.1	86
Szymanski, Bolek	13.1	245
Takefuji, Yoshiyasu	14C.1	316
Taylor, Fred J.	3.2	48
Testard-Vaillant, F.	14B.6	307
Toy, E.J.	14A.3	276
Tracz, William J.	9.1	171
Waksman, Abraham	14C.3	323
Wallace, C.S.	14B.2	293
Whinston, A.	8.1	147
Wise, David S.	15B.6	357
Wittle, Larry D.	10.1	195
Wittle, M.E.	15A.3	336
Yu, Clement	12.1	220
Zwaenepoel, Willy	10.3	208

END

11-87

DTIC